VŠB – Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

# Bakalářská práce

2019

Jakub Pařez

VŠB – Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra kybernetiky a biomedicínského inženýrství

# Absolvování individuální odborné praxe

# Individual Professional Practice in the Company

2019 Jakub Pařez

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra kybernetiky a biomedicínského inženýrství

# Zadání bakalářské práce

Student: **Jakub Pařez**

Studijní program: B2660 Počítačové systémy pro průmysl 21. století

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: ATEsystem s.r.o.
2. Struktura závěrečné zprávy:
   a. Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b. Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c. Zvolený postup řešení zadaných úkolů.
   d. Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e. Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f. Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vedl odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **doc. Ing. Petr Bilík, Ph.D.**

Konzultant bakalářské práce: Ing. Josef Navrátil

Datum zadání: 01.09.2018
Datum odevzdání: 30.04.2019


doc. Ing. Jiří Koziorek, Ph.D.
*vedoucí katedry*

prof. Ing. Pavel Brandštetter, CSc.
*děkan fakulty*

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29.4.2019

.....

Jakub Pařez

*student*

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.


V Ostravě 29.4.2019

.....

Ing. Michal Harhaj

*výkonný ředitel*

Rád bych poděkoval doc. Petru Bilíkovi za pomoc při vypracovávání bakalářské práce a Ing. Josefovi Navrátilovi za excelentní odbornou pomoc, věcné připomínky a konzultaci při vytváření této práce včetně možnosti samostatně pracovat a rozvíjet se.

## Abstrakt

Tato bakalářská práce je závěrečnou zprávou shrnující absolvování individuální odborné praxe ve firmě ATEsystem s.r.o. Pojednává převážně o vývoji produktů cílených na průmyslový trh, konkrétně o vývoji firmware v jazyce C, ovladačů v jazyce C++ a aplikací v prostředí LabVIEW. V první části popisuje úvod do dané technické problematiky. Jsou zde vysvětleny základní rozdíly a je poskytnut primární náhled zvlášť na každou kategorii. Druhá část popisuje postup práce při vývoji firmware produktu „Proudový zdroje pro LED moduly“ v C, ovladače produktu „Řídící jednotka pro objektivy s krokovými motory“ v C++, nástroje pro správu „Proudových zdrojů pro LED moduly“ v LabVIEW, testeru animovaného blinkru světlometu osobních automobilů v LabVIEW, obslužné aplikace „Řídící jednotky pro objektivy s krokovými motory“ v LabVIEW a *benchmarku* grafických a video formátů v LabVIEW. Na závěr je popsáno zátěžové testování zmíněných produktů.

**Klíčová slova:** odborná praxe, vývoj, software, firmware, ovladač, C, C++, LabVIEW, proudový zdroj, LED, řídící jednotka, P-Iris objektiv, animovaný blinkr, automobilový průmysl, obslužná aplikace, benchmark, kodek, zátěžový tester.

## Abstract

This bachelor thesis is a final report summarizing my individual professional practice in ATEsystem, s.r.o. It mainly deals with an industrial product development process, more specifically an embedded firmware development in C language, a driver development in C++ language, and application programming in the LabVIEW environment. The first part focuses on a basic introduction of the technical background. Main differences are explained, and essential information is given for each section. The second part describes a workflow on developing firmware for product “Current source for LED modules” in C, a driver for product “Control unit for lenses with stepper motors” in C++, a tool for management of “Current sources for LED modules” in LabVIEW, a tester of an automotive sequential turn signal lights in LabVIEW, a control application for “Control unit for lenses with stepper motors” in LabVIEW and a *benchmark* of graphical and video formats in LabVIEW. Ultimately, a stress testing of mentioned products is depicted.

**Key words:** professional practice, development, software, firmware, driver, C, C++, LabVIEW, current source, LED, control unit, P-Iris lens, sequential turn signal light, automotive, control application, benchmark, codec, stress tester.

# Obsah

## Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| .NET | – | softwarová platforma firmy Microsoft |
| A | – | statická knihovna v operačních systémech Linux |
| ABI | – | Application Binary Interface – aplikační binární rozhraní |
| amd64 | – | dnes nejrozšířenější 64bitová architektura PC firmy AMD (x86-64) |
| ANSI | – | American National Standards Institute – americká standardizační organizace |
| API | – | Application Programming Interface – rozhraní pro programování aplikace |
| ARM | – | Advanced RISC Machine – populární architektura CPU |
| ASCII | – | American Standard Code for Information Interchange – americký standardní kód pro výměnu informací |
| ASIX | – | česká firma specializující se na výrobu vývojových prostředků |
| ASM | – | assembler – jazyk symbolických adres |
| AVR | – | architektura CPU firmy Atmel |
| BIOS | – | Basic Input-Output System – standard pro rozhraní firmware PC |
| BMP | – | formát rastrové grafiky reprezentující obraz tvořený bitmapou |
| BOOTP | – | Bootstrap Protocol – síťový protokol 7. vrstvy předchůdce DHCP |
| BSOD | – | Blue Screen Of Death – závažné chybové hlášení operačních systémech Windows, ze kterého se lze zotavit pouze restartem |
| CDECL | – | C declaration – volací konvence, která je součástí ABI |
| COM | – | Component Object Model – jazykově neutrální standard binárního rozhraní aplikací firmy Microsoft |
| CPU | – | Central Processing Unit – centrální procesorová jednotka |
| CR | – | Carriage Return – netisknutelný řídící znak z ASCII tabulky, který nastaví kurzor na začátek řádku (hexadecimálně 0x0D) |
| DC | – | Direct Current – stejnosměrný proud |
| DDoS | – | Distributed Denial of Service – distribuovaný DoS útok na dostupnost služby pomocí zahlcení požadavky |
| DHCP | – | Dynamic Host Configuration Protocol – síťový protokol 7. vrstvy který slouží k automatické konfiguraci připojených zařízení |
| DLL | – | Dynamic Link Library – dynamická knihovna v operačních systémech Windows |
| DNS | – | Domain Name System – systém doménových jmen |
| ELF | – | Executable and Linkable Format – standardní formát spustitelných souborů, objektového kódu a sdílených knihoven |
| Ext4 | – | Fourth Extended Filesystem – souborový systém v OS Linux |
| FEI | – | Fakulta elektrotechniky a informatiky |
| FOSS | – | Free and Open-Source Software – svobodný a otevřený software |

| | | |
|---|---|---|
| FPGA | – | Field Programmable Gate Array – programovatelné hradlové pole |
| FPS | – | Frames per Second – snímková frekvence |
| FW | – | Firmware – speciální typ softwaru |
| GC | – | Garbage Collection – automatická správa paměti |
| GCC | – | GNU Compiler Collection – sada překladačů z projektu GNU |
| GUI | – | Graphical User Interface – grafické uživatelské rozhraní |
| GenICam | – | standard pro generické API průmyslových kamer |
| HMI | – | Human Machine Interface – rozhraní mezi zařízením a člověkem |
| HTTPS | – | Hypertext Transfer Protocol Secure – síťový protokol 7. vrstvy, který kombinuje HTTP a TLS k bezpečné komunikaci |
| HW | – | Hardware – fyzické části počítače, protiklad softwaru |
| i.MX | – | rodina MCU firmy Freescale Semiconductors (nyní NXP) |
| ICMP | – | Internet Control Message Protocol – síťový protokol 3. vrstvy pro zasílání informativních a chybových zpráv |
| IDE | – | Integrated Development Environment – vývojové prostředí, které slouží pro tvorbu software |
| IMAQdx | – | Image Acquisition – softwarový modul firmy NI pro prostředí LabVIEW, který umožňuje tvorbu aplikací pro strojové vidění |
| iOS | – | iPhone OS – mobilní operační systém firmy Apple |
| IP | – | Internet Protocol – základní síťový protokol 3. vrstvy |
| IPC | – | Industrial PC – počítač určen pro průmyslové použití |
| IPSW | – | iPhone Software – formát souboru firmy Apple skládající se z různého software a firmware určen pro mobilní zařízení |
| IR | – | Infrared Radiation – infračervené záření |
| ISR | – | Interrupt Service Routine – část kódu (funkce), která se provede v rámci obsluhy přerušení |
| JAI | – | výrobce průmyslových kamer |
| JPEG | – | Joint Photographic Experts Group – standardní metoda ztrátové komprese digitální grafiky |
| JTAG | – | Joint Test Action Group – standard pro testování plošných spojů a programování a ladění čipů |
| KEXT | – | Kernel Extension – formát ovladačů firmy Apple v macOS |
| KMDF | – | Kernel-Mode Driver Framework – softwarový nástroj firmy Microsoft pro vývoj ovladačů běžících v režimu jádra |
| KO | – | Kernel Object – modul jádra operačního systému Linux reprezentovaný objektovým souborem ve formátu ELF |
| LAN | – | Local Area Network – počítačová síť malého rozsahu |
| LED | – | Light-Emitting Diode – elektroluminiscenční dioda |
| LF | – | Line Feed – netisknutelný řídící znak z ASCII tabulky, který posune kurzor na další řádek (hexadecimálně 0x0A) |

| | | |
|---|---|---|
| LIB | – | statická knihovna v operačních systémech Windows |
| LabVIEW | – | Laboratory Virtual Instrument Engineering Workbench – platforma, vývojové prostředí a programovací jazyk firmy NI |
| MAC | – | Media Access Control – spolu s LLC podvrstvou tvoří 2. vrstvu LAN sítí, zahrnuje např. Ethernet, Wi-Fi, DSL nebo FDDI |
| MCU | – | Microcontroller Unit – monolitický integrovaný obvod tvořící jednočipový počítač |
| MISRA C | – | Motor Industry Software Reliability Association C – standard pro vývoj softwaru v jazyce C používaný v automobilovém průmyslu |
| MLC | – | Multi-Level Cell – paměťový prvek schopný ukládat 2 bity v jedné buňce |
| MSP430 | – | 16bitová architektura CPU firmy Texas Instruments |
| NASA | – | National Aeronautics and Space Administration – Národní úřad pro letectví a kosmonautiku |
| NI | – | National Instruments – americká softwarová a hardwarová firma |
| NTC | – | Negative Temperature Coefficient – negastor (termistor – součástka, jejíž odpor závisí na teplotě) |
| NTFS | – | New Technology File System – souborový systém firmy Microsoft |
| O | – | objektový soubor v operačním systému Linux – mezičlánek kompilace a linkování nebo samostatná sdílená knihovna |
| OOP | – | Object-Oriented Programming – objektově orientované programování |
| OS | – | Operating System – operační systém |
| PC | – | Personal Computer – osobní počítač |
| PCB | – | Printed Circuit Board – deska plošných spojů |
| PDF | – | Portable Document Format – formát firmy Adobe pro ukládání textu a obrázků |
| PID (regulátor) | – | Proportional–Integral–Derivative controller – regulátor, který slouží k řízení soustavy v uzavřené smyčce |
| PLC | – | Programmable Logic Controller – programovatelný logický automat |
| PNG | – | Portable Network Graphics – formát rastrové grafiky, který podporuje bezztrátovou kompresi |
| QMH | – | Queued Message Handler – návrhový vzor v LabVIEW |
| RGB | – | Red Green Blue – model aditivního způsobu míchaní barev |
| RS232 | – | Recommended Standard 232 – standard pro sériovou komunikaci |
| RTOS | – | Real-Time Operating System – operační systém reálného času, poskytuje uživateli záruku dokončení činnosti v daném čase |
| RUP | – | Rational Unified Process – iterativní metodologie vývoje softwaru |
| SI (soustava) | – | Le Système International d'Unités – mezinárodní systém jednotek |

| | | |
|---|---|---|
| SO | – | Shared Object – dynamická (sdílená) knihovna v operačních systémech Linux |
| SoC | – | System on Chip – integrovaný obvod zahrnující veškeré komponenty počítače do jediného čipu |
| SREC | – | S-Record – souborový formát firmy Motorola, který reprezentuje binární kód pomocí ASCII znaků, používaný pro programování |
| SSD | – | Solid-State Drive – pevný disk složen z integrovaných obvodů |
| STDCALL | – | volací konvence, která je součástí ABI, používaná Win32 API |
| SW | – | Software – sada počítačových programů, protiklad hardwaru |
| SYS | – | značení ovladačů operačního systému Windows pracujících v režimu jádra, ve skutečnosti se jedná o DLL |
| TCP | – | Transmission Control Protocol – síťový protokol 4. vrstvy, který se vyznačuje obousměrným spolehlivým spojením |
| TIFF | – | Tagged Image File Format – formát rastrové grafiky poskytující kromě obrazu i informace v podobě tagů |
| TLS | – | Transport Layer Security – kryptografický protokol, nástupce SSL |
| TriCore | – | 32bitová architektura CPU firmy Infineon |
| UART | – | Universal Synchronous Asynchronous Receiver and Transmitter – |
| UDP | – | HW periferie, která zajišťuje sériový přenos dat |
| UEFI | – | Unified Extensible Firmware Interface – standard SW rozhraní mezi firmware PC a operačním systémem, nástupce BIOS |
| UIO | – | Userspace I/O – softwarový nástroj pro vývoj ovladačů běžících v režimu uživatele v operačním systému Linux |
| UMDF | – | User-Mode Driver Framework – softwarový nástroj firmy Microsoft pro vývoj ovladačů běžících v režimu uživatele |
| UML | – | Unified Modeling Language – modelovací jazyk, který se v softwarovém inženýrství používá pro návrh a popis systému |
| USB | – | Universal Serial Bus – standard pro sériovou komunikaci |
| UTP | – | Unshielded Twisted Pair – nestíněná kroucená dvojlinka |
| VFIO | – | Virtual Function I/O – softwarový nástroj pro vývoj ovladačů v operačním systému Linux |
| WDF | – | Windows Driver Frameworks – sada softwarových nástrojů firmy Microsoft pro vývoj ovladačů, zahrnuje KMDF a UMDF |
| WDM | – | Windows Driver Models – sada softwarových nástrojů firmy Microsoft pro vývoj ovladačů |
| x86 | – | 32bitová architektura PC firmy Intel, předchůdce amd64 |
| XNU | – | X is Not UNIX – hybridní jádro operačního systému firmy NeXT (nyní Apple) založeném na mikrojádře Mach |

# Seznam obrázků

# 1 Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta

## 1.1 Popis odborného zaměření firmy ATEsystem, s.r.o

Firma ATEsystem, s.r.o. byla založena v roce 2013 a působí na trhu především v oblasti strojového vidění. Od svého počátku se ze skupinky specialistů s mnohaletými zkušenostmi za pět let stala stabilní společnost s více než třiceti zaměstnanci, která se prosazuje třemi hlavními směry, a to systémy pro vizuální inspekci na klíč, dále oficiální distribucí kamerových komponent pro strojové vidění a v neposlední řadě vývojem vlastních produktů. Kromě toho nabízí komplexní odborné poradenství, studie proveditelnosti, testování a může se jako jediná firma v ČR chlubit oceněním NI Silver Alliance Partner s prestižní certifikací NI Vision Specialty.

Pod systémem pro vizuální inspekci si lze představit zařízení střední až větší velikosti vykonávající automatickou vizuální kontrolu, případně i seřízení a montáž, v průmyslovém prostředí. V případě ATEsystem jde převážně o oblast automobilového průmyslu. Testery se skládájí z hlavní mechanické konstrukce, vhodného hardwaru (průmyslové kamery, IPC, PLC, HMI, vlastní produkty firmy – „Proudový zdroj pro LED moduly“, senzory) a softwaru, který spojuje tyto HW prvky do jednoho celku a umožňuje jeho hlavní funkci. SW pro podobné testery je výhodné a dnes v ČR již velmi časté psát v grafickém programovacím jazyce LabVIEW, kdy se v případě zmíněných inspekčních systémů jedná o *know-how* firmy.

Oddělení kamer a komponent ATEsystem nabízí široký sortiment produktů pro strojové vidění od průmyslových barevných, černobílých, řádkových nebo 3D kamer s rozhraním GigE, USB 3.0 nebo Firewire značek Basler, JAI a dalších, přes spektrum objektivů s různou ohniskovou vzdáleností, světelností a různým konstrukčním provedením až k osvětlovacím systémům a kamerovým rozhraním. Kromě prodeje, oddělení také nabízí návrh vhodných kamerových komponent pro libovolnou aplikaci, testování vzorků od zákazníků, studie proveditelnosti a komplexní poradenství v oboru.

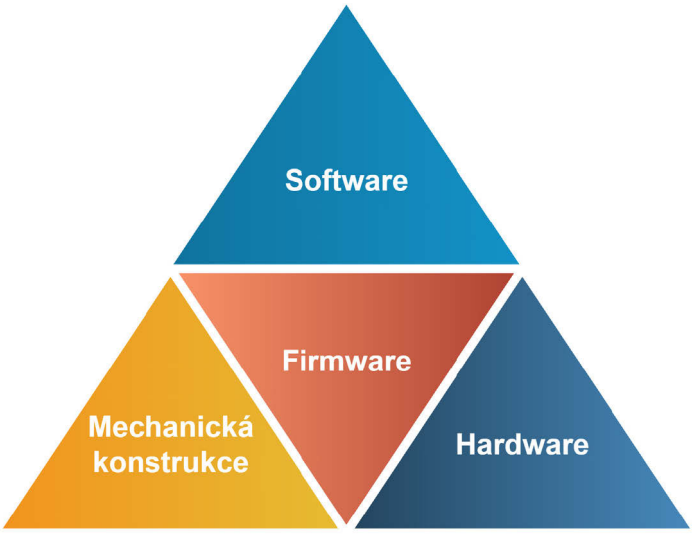
Oddělení vývoje ATEsystem zaměstnává inženýry, kteří vyvíjejí a servisují produkty, jednak chybějící na trhu, tak doplňující nabídku kolegů kamerových specialistů a také hrající nedílnou roli ve výsledku práce kolegů systémových integrátorů. Nejžádanějším produktem je už dlouhodobě „Proudový zdroj pro LED moduly“ s rozhraním Ethernet, který vyniká svými HW i SW parametry, proto je i často dodáván jako součást výše zmíněných systémů pro průmyslovou vizuální inspekci (například jde o kontrolu LED světlometů osobních automobilů). Další produkt, „Řídící jednotka pro objektivy s krokovými motory“, slouží jako mezičlánek pro spojení kamer Basler a objektivů typu P-Iris. Nachází uplatnění v dopravních aplikacích a všude tam, kde jsou vyžadovány pokročilé optické a obrazové vlastnosti a zároveň je potřeba

vzdálené správy a administrace. Velmi podobné cílení má i produkt „Canon Controller“, který ovšem slouží jako mezičlánek kamer Basler a objektivů Canon. Všechny tyto i jiné produkty ATEsystem zahrnují kromě HW i SW podporu ve formě ovladačů, knihoven, obslužných aplikací a příkladů v jazycích C, C++, C# a LabVIEW.

## 1.2 Popis pracovního zařazení studenta

Mé působení ve společnosti ATEsystem během bakalářské praxe je vhodné popsat stručným odstavcem než prostým výčtem úkolů. Primárně jsem pracoval v oddělení vývoje vlastních produktů, kde jsem měl možnost aktivně se účastnit každé části procesu vývoje nového HW/SW výrobku, od definice zákaznických požadavků, analýzy dat, návrhu architektury, výroby prototypu, tvorby dokumentace, implementace FW a SW a testování po nasazení u zákazníka a servisování vadných kusů. V této práci budu psát především o SW, respektive FW, části mého působení ve firmě, protože jak z hlediska stráveného času, tak celkového přínosu jí patří zcela jistě prvenství. Výsledky mé práce lze nalézt na webové stránce zaměstnavatele (spustitelné soubory, instalátory, FOSS projekty, dokumentace, manuály) nebo také přímo v produktech společnosti ATEsystem jako FW zařízení či v systémech pro vizuální inspekci v některé z českých továren. Souhrnně se tedy jednalo o vývoj SW a FW (viz Obr. 1):

- vývoj firmware v C
- vývoj ovladačů v C++
- vývoj aplikací v LabVIEW
- testování

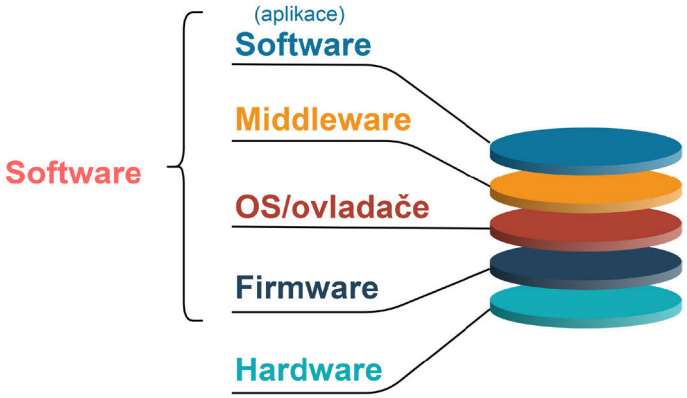



Obr. 1: Základní rozdělení částí vývoje produktů

Nejde o náhodné seřazení typů činností. Popisují hierarchicky úseky SW, resp. FW části vývoje produktů kategorie elektrotechniky a informatiky. Ne u všech produktů je nutná kompletní SW podpora, nicméně dnes už začíná být standardem poskytovat širokou škálu podpůrných aplikací, knihoven a ovladačů, a to nejen z důvodu konkurenčního boje, ale především proto, že svět rychle směřuje od primitivních jednoúčelových nástrojů psaných v jazyce symbolických adres (dnes i v budoucnu má nekorektně jmenovaný *assembler* stále nezastupitelné místo) k vysoce sofistikovaným systémům s mnoha vrstvy abstrakce, které mohou vystupovat v různých rolích, spojovat se do sítí, samy se aktualizovat a sloužit tak uživateli daleko lépe, a to vše samozřejmě vyžaduje komplexní softwarovou podporu. V následujících odstavcích popíši zásadní rozdíly mezi firmwarem, ovladačem a obslužnou aplikací, představím produkty, kterých se to týká, nastíním postup řešení při jejich vývoji, vysvětlím, proč byl zvolen konkrétní programovací jazyk a na závěr zmíním, jak probíhalo zátěžové testování zmíněných produktů.

### 1.2.1 Vývoj firmware v jazyce C

Jelikož následující text bude převážně o určitých převzatých slovech, mezi kterými v laické (někdy i odborné) společnosti panuje značné zmatení, je to třeba hned uvést na pravou míru. Jedná se o firmware, software, middleware, hardware a driver (dále už jen ovladač). Stručně řečeno, firmware, middleware a ovladač jsou software, tedy sada instrukcí spustitelných na CPU vykonávající nějakou akci, na druhou stranu pak hardware je třeba CPU nebo jiná fyzická komponenta zařízení (slangově řečeno *železo*, korespondující s českým překladem anglického *hardware store* – železářství). Definice termínů firmware, middleware a ovladač je už poněkud složitější (viz Obr. 2)




Obr. 2: Schématické vyjádření termínů Software a Hardware

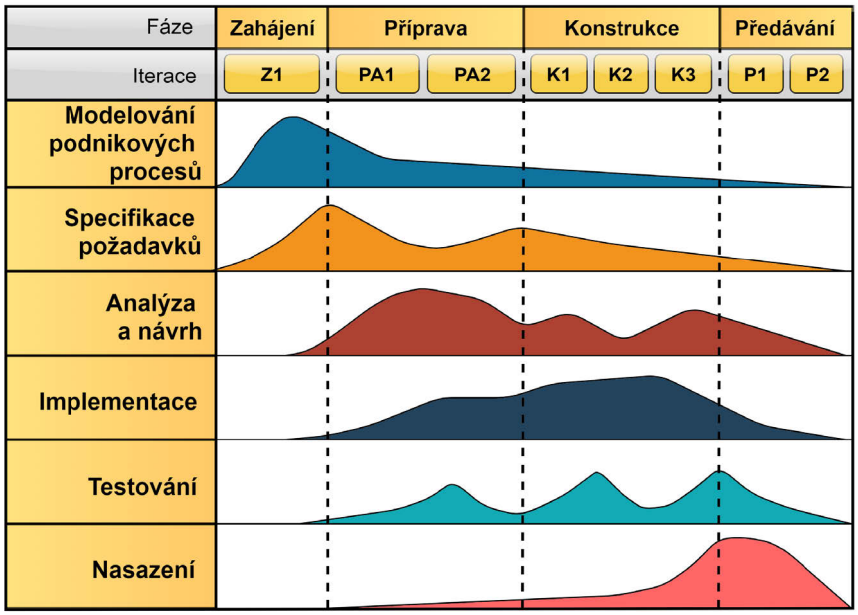
**Firmware** je software, který leží nejblíže hardwaru, je vysoce specializovaný pro daný konkrétní typ hardwaru a běžný uživatel o něho většinou nezavadí. Většinou platí, že firmware je software pro MCU různých architektur (AVR, ARM, MSP430, 8051, TriCore) v *embedded* (jednočipovém) zařízení. Nejčastěji se píše v jazycích ASM (korektně: jazyk symbolických adres, dále pro zkrácení už jen ASM), C a C++ a bývá u něho kladen důraz na malou velikost, optimalizaci a především bezpečnost. Bohužel ale s bezpečností je realita jinde. Dnes a denně jsme svědky opakujících se týž a těch samých bezpečnostních děr, kdy tím opravdovým *evergreenem* jsou ve FW napevno uložené nešifrované přihlašovací údaje včetně hesel. Často se nejedná o problém, pokud ovšem nejde o certifikovaný kardiostimulátor s bezdrátovým připojením. [1] Firmware nejčastěji řeší nízko úrovňové funkce systému, jako třeba práci s pamětí, řízení periferií nebo obsluhu komunikačního rozhraní, ve světe PC tak můžeme nahlížet na BIOS nebo na UEFI jakožto na firmware. Počítačová myš má v sobě mikrokontrolér, který obsluhuje zcela jistě firmware, tak samo digitální budík, na druhou stranu třeba Apple ne zcela správně nazývá svůj IPSW soubor pro aktualizaci iOS v telefonech iPhone firmwarem, když jde ve skutečnosti o balík OS, podpůrného softwaru, middlewaru a firmwaru.

**Middleware** je na rozdíl od firmwaru více abstraktní pojem, který není příliš masově rozšířen. Jedna se v podstatě o jakýkoliv software (může to být i firmware), který slouží jako spojovací vrstva, rozhraní mezi SW vrstvami, často je také zobecňován na cokoli, co leží mezi jádrem OS a uživatelskou aplikací.

**Ovladač** je software, který poskytuje rozhraní ke konkrétnímu hardwaru. Ovladač může být součástí firmwaru a poskytovat funkce periferie vyšší vrstvě, třeba RTOS, tak samo může být ovladač součástí softwaru PC a poskytovat rozhraní připojeného zařízení vyšší vrstvě OS. Ovladač se většinou blíží popisu firmwaru, kdy jde často o relativně malý, optimalizovaný kód napsaný v ASM/C/C++ řešící nízko úrovňové funkce v blízkosti jádra, není to ale pravidlo. Je zcela v pořádku nazývat kód ovladačem, pokud běží v *user-space* (*ring* 3), není součástí jádra ale uživatelské aplikace, poskytuje-li hlavní rozhraní pro určitý hardware.

Vývoj firmware je vývoj software se vším, co k tomu náleží, kdy se dnes nejčastěji v iteracích cyklicky prochází proces (viz Obr. 3), rozložen do čtyř postupných fází. RUP definuje fáze jako zahájení, přípravu, konstrukci a předávání, není ale problém si je přizpůsobit, kostra procesu by však měla zůstat stejná, přičemž je nutné si uvědomit, že následující činnosti se neprovádějí sekvenčně, ale paralelně s různým procentuálním zastoupením: [2]

1. Modelování podnikových procesů
2. Specifikace požadavků
3. Analýza a návrh architektury
4. Implementace
5. Testování
6. Nasazení




Obr. 3: Vývoj softwaru iteračním způsobem dle procesu RUP

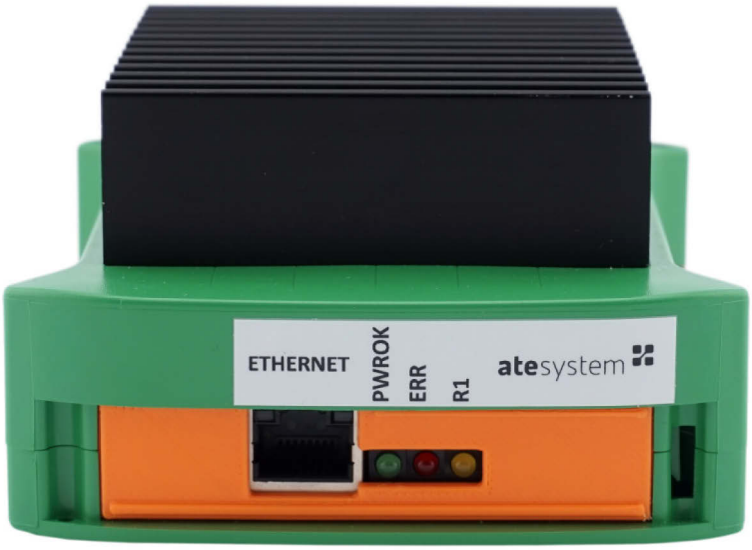
Kromě volby a správy vývojového prostředí, ladících nástrojů, překladače a *linkeru* zde existují jistá specifika, která běžný SW vývojář nezná a z podstaty je ani nepotřebuje znát. Jedná se například o aktivní využívání logického analyzátoru či osciloskopu, bez kterého je práce *embedded* vývojáře těžko realizovatelná. Klíčová je také znalost hardwaru a digitálního designu. Je opravdu nezbytné vědět, jak fungují komponenty mikrokontroléru, periferie a další logické bloky v daném SoC, protože firmware slouží k jejich řízení, a bez znalostí jejich funkčnosti a základních principů fyziky můžou vzniknout i závažné škody, jako například v prosinci 1998, kdy sonda NASA Mars Climate Orbiter v hodnotě 125 milionů dolarů nezvládla na Marsu přistávací manévr. Vyšetřovací komise zjistila, že software v řídícím středisku počítal trajektorii v jednotkách SI ($N \cdot s$), zatímco FW sondy očekával jednotky imperiální ($lb \cdot s$). [3]

V minulosti, když odhlédneme od děrných štítků, dominoval v této oblasti jazyk symbolických adres – *assembler*. Z části se používá i dnes, především na mikro optimalizace a malé kritické kusy kódy (např. *bootloader*), z velké části byl však nahrazen jazykem C, ať už proto, že rozšiřování velkého projektu v ASM je časově i kapacitně zbytečně moc náročné, tak především z důvodu masového nástupu ARM jader s více úrovňovou *pipeline*, kde její optimální naplnění zajistí většinou lépe stroj (překladač) než člověk. V tomto případě ale být nahrazen neznamená být vyřazen. Znalost instrukčních sad a strojových instrukcí v této oblasti velmi často vymezuje pomyslnou čáru mezi kvalitním vývojářem a těmi ostatními. Jazyk C++ je zde už přes 30 let a GCC podporuje C++ jak u AVR, tak ARM i dalších architektur, z řady důvodu ale není (odborná) veřejnost příliš nakloněna používání C++ pro programování firmware. Nejslyšitelnějšími argumenty jsou, že v C++ projektu s mnoha úrovněmi polymorfismu není vždy hned jasné, jaký kus kódu se spustí, a co a kde se alokuje v paměti, což jsou vlastnosti pro tuto oblast nepříjemné. Částečně tohle řeší dialekt EC++ (Embedded C++), který z C++ vypouští často problémové věci jako výjimky, vícenásobnou dědičnost a šablony. Nicméně zkušený programátor s pokročilými znalostmi C a C++ by neměl mít problém s vývojem kvalitního firmware v moderním C++17 za použití současných překladačů. Jistou nevůli psát firmware v C++ tak lze částečně pokládat za opodstatněný mýtus. Jazyk C a jeho evoluce (ANSI C, C99, C11) případně pak MISRA C (standard vyvinut pro automobilový průmysl, který mj. zakazuje ukazatelovou aritmetiku vyjma indexování polí) tak je a nějaký čas i nadále bude nejpopulárnější volbou pro vývoj firmware.

Výsledný FW soubor se nejčastěji distribuuje ve formátech ELF, Intel-HEX nebo SREC a minimálně obsahuje vždy instrukce kódového paměťového segmentu (.text) v binární či ASCII (hexadecimální) podobě. Testování a bezpečnost firmwaru jsou kapitoly samy o sobě. Testování se budu věnovat v kapitole 2.4, kdy zmíním nástroje v LabVIEW, které jsem vyvinul pro testování produktů společnosti ATEsystem, bezpečnost je však tématem, který dnes hýbe světem. Bezpečnost, tím je myšlen proces zabezpečení systému proti zneužití, by měla

být vždy na prvním místě, ať už se jedná o set-top box či defibrilátor, a zabezpečením rozhodně není myšleno *security-by-obscurity*. V prvním případě může dojít k začlenění set-top boxu do *botnetu* a následný DDoS útok může ochromit nemocnici (např. incident v Bostonské dětské nemocnici z roku 2014). Druhý případ potvrzuje zpráva amerického ministerstva vnitřní bezpečnosti z roku 2013, kdy ministerstvo varuje před 300 zdravotnickými produkty od více než 40 firem od infusních pump, přes anestetické systémy po externí defibrilátory a monitorovací stanice pacientů, které mají ve firmwaru napevno uložená nešifrovaná přihlašovací jména a hesla. [4]

Mým úkolem bylo převzít částečně funkční firmware k produktu „Proudový zdroj pro LED moduly“ (viz Obr. 4), přepsat ho do lepšího stavu, doplnit funkčnost, odladit, najít chyby a přidat nové funkce.




Obr. 4: Produkt „Proudový zdroj pro LED moduly“

Sice mi byl ulehčen začátek již existujícím kódem s funkční implementací PID regulátoru, nicméně se nejednalo o čistý, zdokumentovaný a lehce čitelný kód, strávil jsem tedy delší dobu jeho analýzou a následným přepsáním zhruba poloviny modulů dle mých možností a schopností. Následovalo velmi dlouhé, ale úspěšné hledání chyby, která se při zátěžových testech projevovala zhruba jednou za 6–12 hodin. Pokračoval jsem přidáním nových funkcí (vzdálená aktualizace firmwaru, podpora verze produktu s nízkým rozsahem výstupního proudu a další), s průběžným testováním zpětné kompatibility a zátěžovým testováním. Detaily jsou popsány v kapitole 2.1.1.

### 1.2.2 Vývoj ovladačů v jazyce C++

Pro ovladače platí stejné obecné principy, které jsem zmínil v kapitole 1.2.1. Začíná se specifikací požadavků, pokračuje analýzou a návrhem architektury přes implementaci a testování až po nasazení. Vývoj ovladačů realizuje nejčastěji firma, která vyrobila konkrétní hardware, pro který je ovladač určen. Úspěšný vývoj následuje zpravidla certifikací a integrací u výrobců operačních systémů do infrastruktury OS. U ovladačů je podobně jako u firmwaru kladen vysoký důraz na spolehlivost, protože často komunikují přímo s jádrem OS a jakákoli chyba tak většinou způsobí pád celého systému. Mezi uživateli Windows je známá BSOD zobrazující Ntfs.sys (ovladač souborového systému NTFS) jako zdroj problému, analogicky v OS Linux existuje *EXT4-fs kernel panic* (ovladač souborového systému Ext4 je součástí jádra). Příčinou může být zakázaná operace s pamětí, nekompatibilní hardware nebo poškozený či chybový ovladač. Jiné varianty BSOD nebo *kernel panic* jsou také často vázány na podobné problémy s ovladači.

Ovladač může být myšlen v kontextu firmwaru (např. I2C ovladač pro MCU rodiny i.MX) nebo v kontextu softwaru (např. již zmíněný Ntfs.sys). Následně bude uveden zjednodušený přehled možností vývoje ovladačů pro tři dominantní OS:

**Windows**

Ačkoli je jádro Windows napsáno v jazyce C, většina Win32 API je v C++ a stejně tak pro psaní ovladačů doporučuje Microsoft C++. Od verze 2000 zde existuje WDF (Windows Driver Frameworks), sada knihoven a nástrojů, která lze díky IDE Visual Studio jednoduše použít pro tvorbu ovladačů. WDF ve skutečnosti funguje jako abstrakce nad technologií WDM (Windows Driver Model), která pochází z verze 98 a která představuje univerzálnější, byť komplikovanější, nástroj pro tvorbu Windows ovladačů.

- WDM (*Kernel*, používán každým Win. ovladačem, DLL, přípona SYS)
- WDF (*framework* usnadňující vývoj díky abstrahování WDM)
  - KMDF (*Kernel*, *framework* WDF, typ souboru DLL, přípona SYS)
  - UMDF (*User-space*, *framework* WDF, COM, typ souboru DLL)

**Linux**

Linux kernel je oproti hybridnímu Windows jádru monolitický a celý napsaný v C, je tedy standardní psát ovladače pro Linux v jazyce C. Je nutné buď pokaždé znovu jádro zkompilovat nebo použít kernel modul. Možností je také UIO nebo VFIO.

- součást jádra (*Kernel*, je nutno vždy rekompilovat celé jádro)
- kernel modul (*Kernel*, bez kompilace jádra, typ souboru O/KO)
- UIO/VFIO (*User-space*, *framework* usnadňující vývoj)

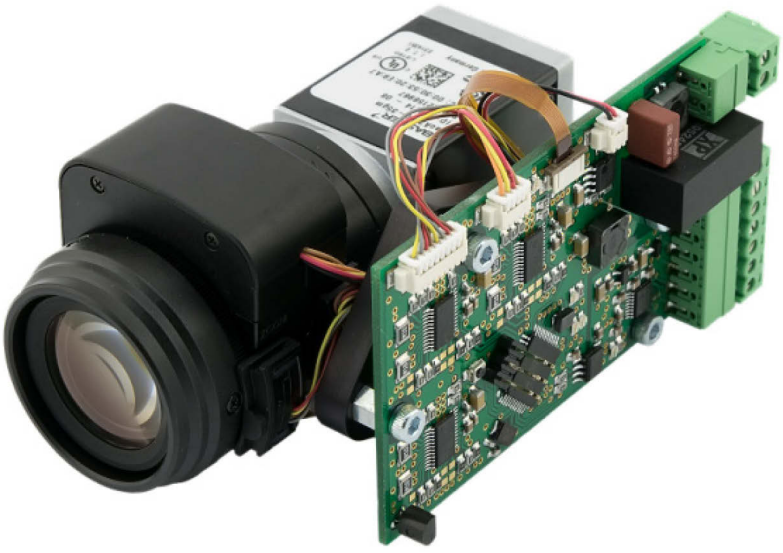
**macOS**

Systém macOS vychází ze systému Darwin a má hybridní jádro XNU (Mach + BSD), které vzniklo jako FOSS několik let před tím (1989), než Linus Torvalds začal psát linuxové jádro (1991). Není tedy pravda, že systémy macOS a Linux mají podobné jádro. Apple nad jádrem XNU vytvořil mnoho proprietárního kódu včetně *frameworku* I/O Kit pro ovladače, ve kterém jako jediný velký vývojář adoptoval dialekt C++ jazyk EC++. Klasickou cestou pro tvorbu macOS ovladačů ale stále zůstává jazyk C.

- generické rozšíření kernelu (*Kernel*, jazyk C, typ souboru KEXT)
- I/O Kit *framework* (*Kernel/User-space*, jazyk EC++, typ KEXT)

Výše zmíněné principy jsou tím pravým způsobem vývoje systémových ovladačů pro majoritně zastoupené OS, neznamená to ale, že všechno ostatní, co do této kategorie nespadá, nelze nazývat ovladačem. Je v pořádku nazývat obyčejnou dynamickou či statickou knihovnu ovladačem, pokud poskytuje primární rozhraní k určitému hardwaru.

Mým úkolem bylo vytvořit ovladač pro produkt „Řídící jednotka pro objektivy s krokovými motory“ (viz Obr. 5). Tento produkt je čerstvým přírůstkem do portfolia společnosti ATEsystem. Aby si firma zajistila dobrou pozici na trhu v oblasti průmyslových a dopravních aplikacích strojového vidění, je nutné poskytnou zákazníkům více než konkurence, minimálně však kvalitní SW podporu.



Obr. 5: Produkt „Řídící jednotka pro objektivy s krokovými motory“

Díky analýze trhu mi bylo sděleno, že se mám zaměřit pouze na OS rodiny Windows a Linux. Jako programovací jazyk jsem zvolil C++17, protože platforma Windows byla dle zákaznických preferencí na prvním místě a jako *framework* UMDF spolu s klasickou dynamickou knihovnou (ve formátu DLL). Podpora Linuxu měla být řešena první pouze formou knihoven (ve formátu A/SO) a později případně přepsána do formy UIO nebo kernel modulu, v závislosti na poptávce zákazníků. V průběhu implementační fáze se však objevil SW problém, který ani já ani vedení nečekalo, muselo se tak přejít zpět k analýze a k návrhu architektury, kdy se následně musela změnit celá koncepce. Ovladač se nakonec vyvinout podařilo, nikoli ale pomocí UMDF formou DLL, ale ve formě open-source knihovny a příkladů, kdy si zákazník podle potřeby zkompiluje svou aplikaci spolu s ovladačem sám. Detaily vysvětlím v kapitole 2.2.1.

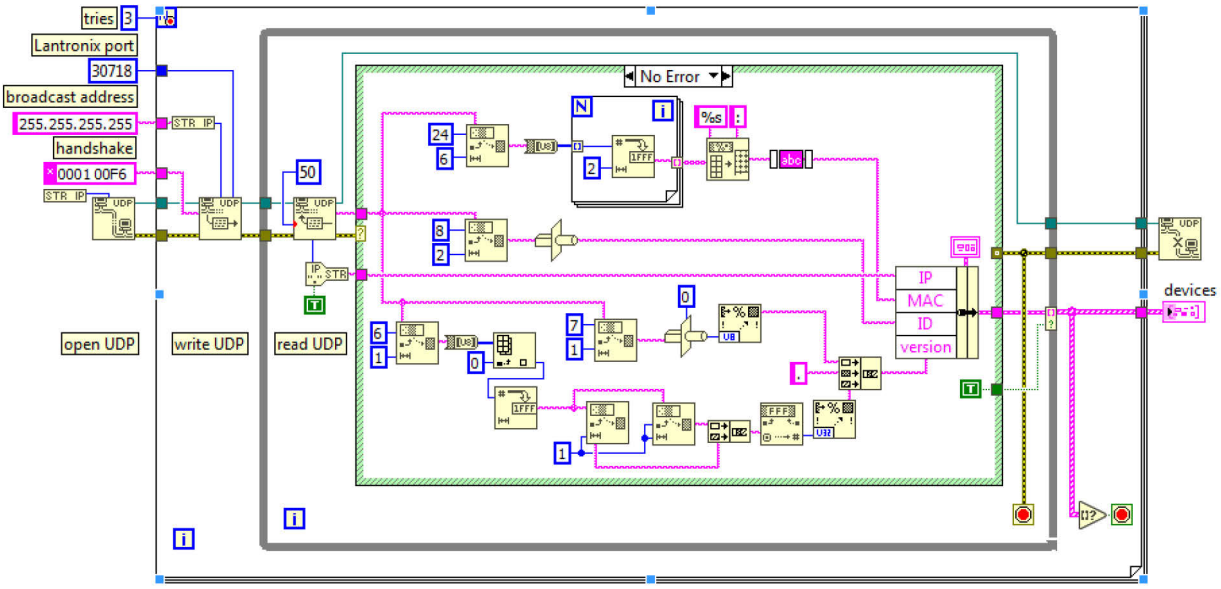
### 1.2.3 Vývoj aplikací v prostředí LabVIEW

Vývoj aplikací je příliš obecný termín na popsání postupu jejich vývoje, jedná se o software, kdy opět platí již zmíněné principy, zaměřím se proto na výběr jazyka. LabVIEW je proprietární vývojové prostředí, sada nástrojů a knihoven pro grafický programovací jazyk G. Vzniklo více než před 30 lety, jeho tvůrce texaská firma National Instruments ho napsala v převážně v jazyce C++ a dnes je z něj široce rozšířený multiplatformní nástroj, integrující .NET Framework a mnoho dalších modulů jako moduly pro strojové vidění, FPGA, sběr dat, *real-time* aplikace a další. Uplatnění LabVIEW nachází převážně v průmyslu a ve vědeckých laboratořích, kde je často potřeba rychlý zásah do funkčnosti systému za co nejkratší čas. Dále LabVIEW vyniká v paralelismu, který vychází ze základního paradigmatu jazyka G a tím je *dataflow* (tok běhu programu není sekvenční jako u imperativních jazyků, ale řídí se jinými pravidly). Výsledný zkompilovaný program je spustitelný soubor pro konkrétní OS, má uživatelské rozhraní a běží díky virtuálnímu stroji, který musí být na cílovém stroji nainstalován (podobná filozofie jakou má C# nebo Java). Platforma LabVIEW obsahuje nativně ladící i UML nástroje, jako verzovací systém je však lepší použít třeba Git.

Firma ATEsystem s certifikovanými vývojáři od National Instruments používá SW i HW od NI ve všech oddělení, kde většina zaměstnanců jsou zkušenými LabVIEW programátory, někteří vlastní i prestižní ocenění „LabVIEW Certified Architect“, je proto logické, že firemní standardní jazyk pro většinu (GUI) aplikací je právě LabVIEW, a ne celosvětově populární C# nebo Java. Další výhodou LabVIEW je podpora průmyslových kamer s rozhraním GigE a USB 3.0 a také modul pro strojové vidění, který nabízí široké možnosti zpracování obrazu. Nevýhodou je závislost na ekosystému NI, tím je myšlena nutnost přítomnosti SW „LabVIEW Run-Time“ na cílovém zařízení, především pak jeho velikost a zpětná nekompatibilita.

Mým prvním úkolem v kontextu aplikací bylo vytvořit zátěžový tester pro „Proudový zdroj pro LED moduly“, kterému se budu věnovat v kapitole 2.4.1. Během práce na něm a na FW zdroje jsem postrádal možnost v LabVIEW vyhledat všechny připojené proudové zdroje na LAN síti a jejich IP adresy. Musel jsem neustále otevírat vyhledávací aplikaci třetí strany a IP z ní kopírovat. Ve volném čase jsem tak pomocí aplikace Wireshark zanalyzoval komunikační protokol oficiální konfigurační aplikace pro Ethernet/UART převodník proudového zdroje (Lantronix X-Port) a vytvořil dle ní alternativu v LabVIEW. Tento program se pak zalíbil mému nadřízenému a postupem času jsem kolem něj vybudoval obslužnou aplikaci „Device Configurator“, která nyní slouží jako hlavní administrační nástroj pro produkt „Proudový zdroje pro LED moduly“.

Nejdůležitější funkcí je hledaní zdrojů v LAN síti, nastavování síťových parametrů (IP adresa, maska podsítě, DNS server, režimy – DHCP/Auto IP/BOOTP/static), vzdálená aktualizace firmwaru proudového zdroje a integrace s již existující aplikací pro používání konkrétního kusu. Níže (viz Obr. 6) je blokový diagram zmíněné aplikace pro vyhledávání zařízení Lantronix v síti LAN, která využívá paketů UDP *broadcast*.




Obr. 6: Blokový diagram aplikace, která vyhledává zařízení Lantronix v síti LAN

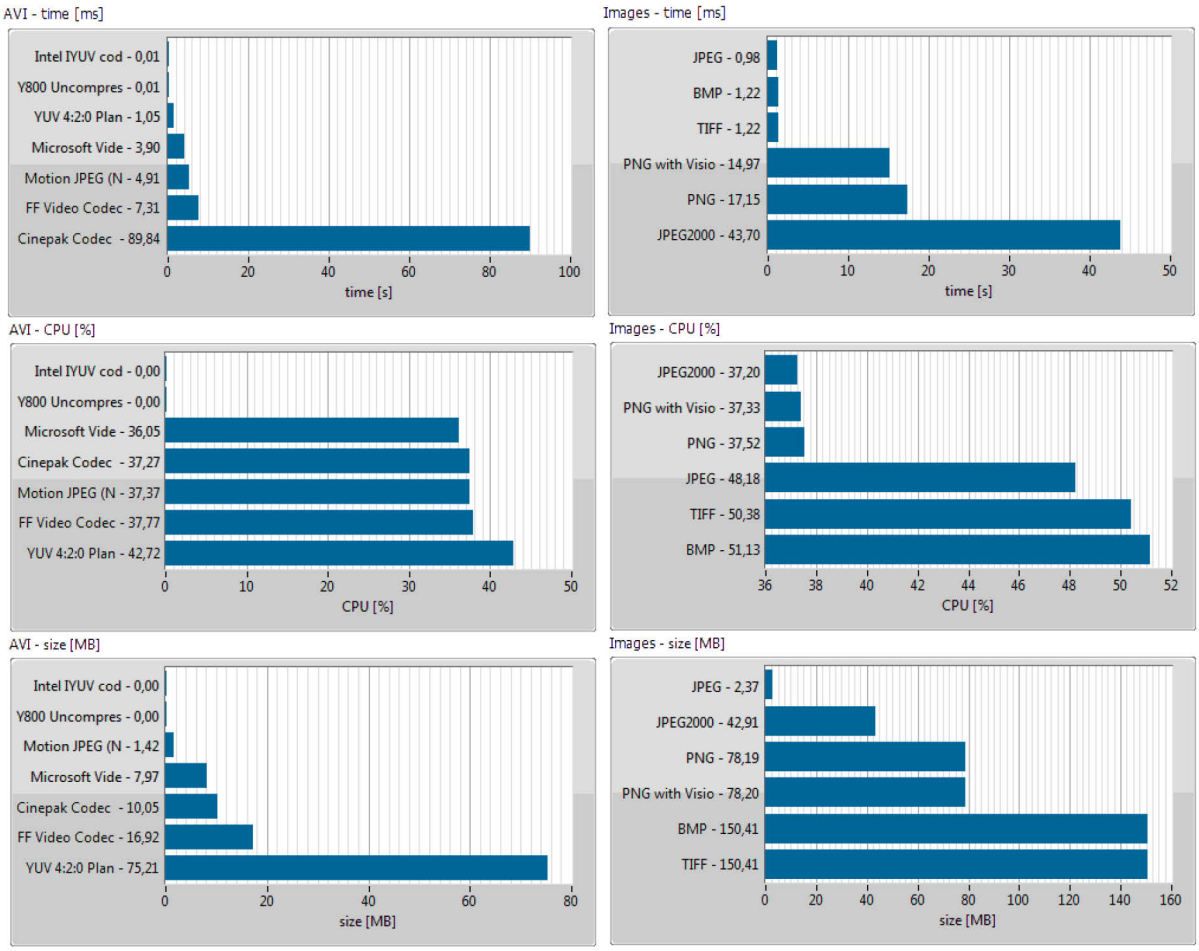
Dále jsem dostal za úkol vyvinout algoritmus pro analýzu animovaného blinkru světlometů osobních automobilů a naimplementovat jej v LabVIEW formou *plug-inu*, který by se stal součástí rozsáhlého firemního systému vizuální inspekce. Dostal jsem na testování před-produkční vzorek LED světlometu osobního automobilu, kameru s Ethernetovým rozhraním Basler a řídící jednotku elektroniky světlometu. Nasnímané vzorky animování blinkru z různých úhlů jsem si uložil na disk a pak postupně navrhl a naimplementoval v LabVIEW algoritmus, který z nasnímaných vzorků vypočetl dobu trvání svícení jednotlivých segmentů blinkru, celkový čas a další parametry. Nejdůležitější bylo detektovat, jestli se segmenty rozsvěcovaly postupně a ve správném pořadí, jestli se rozsvítily všechny a svítily až do konce, a jestli všechny svítily předepsaný časový interval. Intenzitu, barvu světla a další parametry jsem analyzovat neměl, to měl na starosti jiný SW. I přes potíže s kolísáním a nepřesností opakovaně měřených hodnot, způsobených snímáním animace blinkru přes difuzor, který znatelně zhoršoval detekci segmentů, se mi podařilo uspět a dnes je aplikace nasazená u zákazníků ve výrobě. Díky situaci na trhu, kdy se razantně zvedá poptávka po animovaném blinkru, je pravděpodobné, že tato aplikace bude v budoucnu stále více využívána.

Mým dalším úkolem bylo převzít rozdělanou a částečně funkční obslužnou aplikaci pro produkt „Řídící jednotka pro objektivy s krokovými motory“ a pokračovat na jejím vývoji. Jde o stejný produkt, pro který jsem psal ovladač (viz kapitola 2.2.1). Obslužná aplikace s grafickým rozhraním se hodí pro tento typ produktů především během testování, prezentací a rychlého ladění. U zákazníka, tím je myšleno v konkrétním případě například v dopravním odvětví (sloupy dopravní infrastruktury), se používá především ovladač, který poskytuje kompletní rozhraní. V rámci kvalitní SW podpory je ale obslužná aplikace nezpochybnitelným pilířem, který by měl mít standardy srovnatelné s ovladačem. Aplikaci se mi podařilo opravit, přepracoval jsem grafické rozhraní, odstranil chyby vedoucí k nestabilitě a přidal novou funkcionalitu v podobě vyhledávání zařízení, ukládání konfigurace a podpory pro novou verzi firmwaru. Produkt existuje ve dvou variantách, s Ethernetovým a RS232 rozhraním. Z důvodu značné velikosti a dalších problémů nebyla do aplikace přidána podpora pro Ethernetovou verzi, pro kterou tak zůstává ovladač jedinou volbou.

Posledním zde zmíněným úkolem bylo vytvořit aplikaci, která bude sloužit k analýze a porovnání grafických a video formátů v LabVIEW tzv. *benchmark*. Pokud se používá LabVIEW na práci s grafikou nebo videem a je třeba provést konverzi nebo uložení na disk, je velmi důležité zvolit správný formát, resp. *kodek*. Byla stanovena čtyři hlavní kritéria pro posouzení výsledku od nejdůležitějšího po nejméně důležité:

1. Čas – doba trvání konverze a uložení předem definované množiny snímků
2. Kvalita – je měřena jako jediná subjektivně a určuje se v kontrastu s originálem
3. Velikost – výsledná velikost videa či grafických souborů po konverzi a uložení
4. Náročnost – průměrné vytížení všech CPU jader za celou dobu trvání konverze

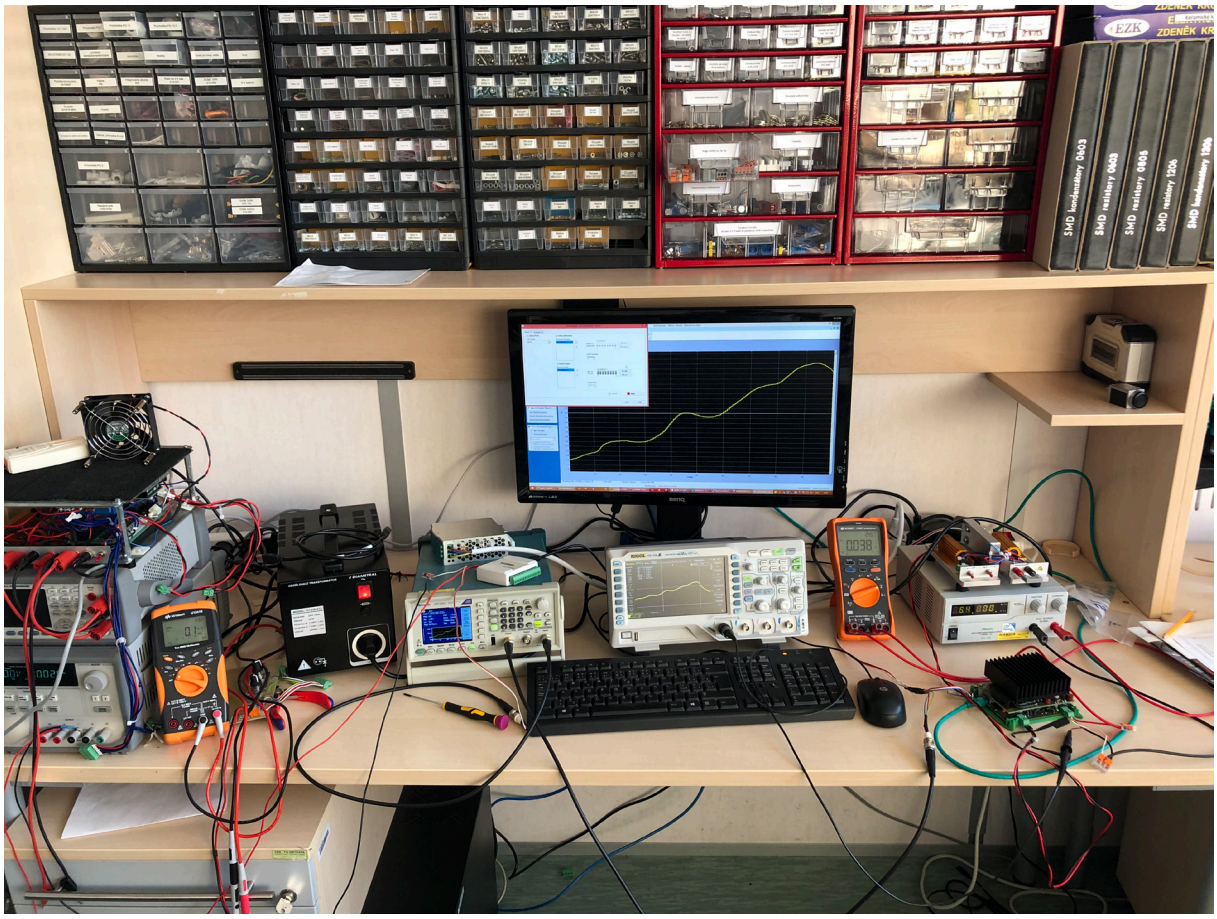
Algoritmus pro aplikaci nebyl nijak složitý, nejprve se nasnímala data v nekomprimovaném formátu kamerou Basler a uložila do operační paměti, následovalo postupné procházení snímků, jejich konverze a uložení na disk. Během procesu se monitorovala velikost, čas a náročnost, které se po ukončení zanesly do sloupcových grafů, zvlášť pro video a zvlášť pro grafické formáty. Nutno poznamenat, že aplikace pracuje pouze s *kodeky* podporovanými v prostředí LabVIEW. Tyto *kodeky* pak musí být ještě podporovány a nainstalovány v testujícím OS, pokud je to splněno, LabVIEW automaticky poskytne jejich seznam. Výsledky (čas, velikost, náročnost) zvlášť pro video a grafické formáty lze vyčíst z grafů (viz Obr. 7), kvalita byla hodnocena subjektivně. Detailněji se tomuto problému věnuji v kapitole 2.3.4.




Obr. 7: Část čelního panelu aplikace *benchmark* grafických a video formátů s výsledky

### 1.2.4 Testování

Testování software je neodmyslitelnou součástí vývoje. Testovat by se mělo v každé fázi vývoje v každé iteraci jeho procesu. Testování je dnes velmi široké odvětví informatiky, zjednodušeně můžeme říci, že sahá od těch, kteří netestují vůbec, přes *unit* testery v korporacích, kteří testují jednoduše a za pomocí populárních *frameworků*, *unit* testery v automobilovém průmyslu, kde je nutné dodržovat přísné normy a testovat jen dle UML diagramů, až po některé americké vládní instituce, kde se testuje software pouze za pomocí matematických důkazů. Další kapitolou je pak letectví, kosmonautika a armáda. Platí, že neotestovaný software je horší než žádný software, ale také samozřejmě platí, že testování není zadarmo. Je proto důležité najít v tomto rovnováhu, ujasnit si, do jaké kategorie testovaný produkt patří a do jaké míry se vyplatí investovat do testování, a tomu pak přizpůsobit vývoj.



Obr. 8: Přístrojové vybavení použité při zátěžovém testování proudového zdroje

Během práce na produktech „Proudový zdroj pro LED moduly“ a „Řídící jednotka pro objektivy s krokovými motory“ jsem vytvořil několik aplikací v LabVIEW, které slouží k testování, především se jedná o zátěžové testování. Někdy se stane, že klasický *unit* test neodhalí žádnou chybu. K ní pak dojde náhodně jednou za dlouhou periodu nejčastěji ve standardních podmínkách. V tomto případě je třeba simulovat chybové prostředí, využít například LabVIEW, které je pro tyto účely velmi vhodné a provést zátěžový test. V mém případě se jednalo nejčastěji o testování komunikace mezi produktem a PC, a testování funkcionality z pohledu uživatele reprezentované a poskytované částmi FW a HW (viz Obr. 8).

Vytvořil jsem test pro produkt „Proudový zdroj pro LED moduly“, protože došlo na situaci s náhodně vyskytující se chybou v dlouhých intervalech, a úspěšně jsem po následné analýze určil příčinu chyby. Další podobné testy jsem napsal pro produkty „Canon Controller“ a „P-IRIS Controller“, které také pomohly kolegům správně lokalizovat příčinu chybovosti. Poslední zátěžový test nebyl iniciován žádnou chybou, bylo ale nutné ověřit bezchybnost nově přidané funkcionality do FW proudového zdroje (vzdálená aktualizace FW). Test proběhl zhruba stokrát s různými parametry pro různé produktové verze a přidaná funkcionalita tak byla vyhodnocena jako nezávadná.

# 2 Seznam úkolů zadaných studentovi v průběhu odborné praxe a zvolený postup řešení

## 2.1 Vývoj firmware v jazyce C

### 2.1.1 Firmware „Proudového zdroje pro LED moduly“

„Proudový zdroj pro LED moduly“ je produkt firmy ATEsystem (viz Obr. 9), primárně sloužící k napájení a testování LED modulů a řetězců, které neobsahují vlastní napájecí elektroniku. V zájmu zachování firemního IP, následující text bude obsahovat pouze veřejně publikovatelné informace v kontextu tohoto produktu. Přes své malé rozměry se jedná o komplikované a sofistikované zařízení. Napájecí napětí 24 V DC ±10 % a maximální příkon 150 W umožňují všestranné použití v průmyslovém prostředí, do kterého je tento produkt cílen. Z napájecího napětí je nejprve spínáním vytvořeno napětí vyšší, které je následně lineárně regulováno pomocí PID regulace na výsledné (2–50 V). Díky tomuto systému je možné mít zařízení o rozměrech 101 x 120 x 62 mm s výstupním zvlněním proudu o hodnotách nižších než 10 mA. Rozsah proudového výstupu závisí na verzi produktu, verze Standard má rozsah 100 mA – 2000 mA, verze Low-Range 20 mA – 350 mA. Přesnost nastavení proudu je dle specifikace maximálně ±3 %, nicméně u novějších verzích firmwaru se s přesností dostáváme pod 1 %. Komunikace zdroje je řešena buď pomocí RS232 nebo Ethernetu, což je dnes jednoznačně nejpopulárnější volba. Přes TCP spojení se komunikuje ASCII proprietárním protokolem. Seznam instrukcí čítá více než 70 příkazů a je podrobně popsán v příslušném dokumentu, na kterém jsem v rámci vývoje FW pracoval. [5] [6]


Obr. 9: „Proudový zdroj pro LED moduly“ – pohled na digitální I/O

Kromě regulace proudu se ve zdroji nachází další funkcionality, využitelné během testů LED modulů a automatické inspekce. Jde třeba o měření a porovnávání výstupního napětí vůči předem přednastaveným hodnotám, kdy tak lze rychle identifikovat vadný LED modul. Dále se často zákazníky využívá tzv. *trigger mode*, kdy je zdroj ovládán manuálně pomocí digitálních vstupů a výstupu, nejčastěji z PLC. Zdroj dokáže také měřit NTC a kódovací odpor, což může být vyžito k automatizaci procesů nadřazeným systémem. Ochrany proti přehřátí, přetížení, přepětí a zkratu jsou u průmyslových produktů samozřejmostí, nechybí tak ani zde. Ke zdroji je dodáván LabVIEW ovladač, implementující proprietární ASCII protokol a pak také SW „Device Configurator“, který je mou prací a bude popsán v kapitole 2.3.1. Mým úkolem bylo převzít, zkultivovat, opravit a rozšířit FW o nové funkce zhruba během jednoho měsíce. Vývoj firmware ale nebývá jednorázová činnost, postupem času vznikají nové zákaznické požadavky a HW revize, proto nelze s přesností definovat časový rozsah tohoto úkolu, především také z důvodů stávající i budoucí práce v oddělení vývoje firmy ATEsystem.

První dny jsem pouze analyzoval strukturu, jelikož UML diagramy nebyly k dispozici. Narazil jsem na mnoho překážek, kdy mi většinou pomohlo nastudování technické a vývojové dokumentace nebo použití osciloskopu či logického analyzátoru. Během analýzy jsem také ladil řetězec nástrojů pro vývoj FW. Použitá rodina 8bit MCU je ve světě široce rozšířená, a proto jsem neměl žádný problém s instalací IDE, překladače a *linkeru*. Co se týče *debuggeru*, měl jsem na výběr mezi dvěma oficiálními zařízeními s plnou podporou ladění přes JTAG, bohužel návrh HW s touto variantou nepočítal. Veškeré pokročilé funkce rozhraní JTAG jako třeba krokování a *breakpointy*, sledování hodnot CPU registrů, sledování aktuálního obsahu pamětí nebo *disassembly* tak šly stranou a mě zbývalo ladění pomocí *printf*. Nebylo to nic zásadního, komfort vývoje to ale lehce snížilo.

Po úspěšné analýze jsem přistoupil ke kroku kultivace kódu. Bylo třeba kompletně přepsat hlavní a komunikační modul, projít zbytek a opravit varovné hlášky překladače. Paralelně s tímto procesem jsem musel neustále kontrolovat zachování funkčnosti ve všech směrech, a to stabilitu systému, nulovou chybovost komunikace a přesnost regulace. Proto jsem měl k dispozici rezistorovou zátěž, která simulovala připojený LED modul, dva multimetry, jeden na měření výstupního napětí a druhý na měření proudu, dále prototypovou desku s potenciometry, indikačními LED diody a tlačítky, simulující vstupně-výstupní připojitelné zařízení, a samozřejmě pak laboratorní napájecí zdroj. Postupoval jsem po iteracích, kdy jsem nejprve provedl změnu v kódu, přehrál FW, pak jsem pokračoval ověřením funkčnosti a testováním.

Testování probíhalo pokaždé zhruba stejně a to tak, že jsem postupně podle interního komunikačního manuálu vyzkoušel veškeré příkazy od prvního po poslední včetně nejčastějších kombinací (restart –> identifikace –> nastavení limitů napětí –> nastavení limitů proudu –> nastavení hodnoty proudu –> zapnutí výstupu –> změření veškerých veličin –> změna hodnoty proudu –> změření veličin –> vypnutí výstupu). Pokud se ukázalo, že je vše v pořádku, zaznamenal jsem příslušné změny do dokumentů i do kódu a pokračoval dál. Kromě opravy a přepisování jsem pokračoval dodatečnou dokumentací kódu formou komentářů ve zdrojovém kódu. Správně okomentovaný kód je naprosto nezbytný pro případy, kdy je v něm nutné něco

opravit, kdy je nutné přidat další funkčnost a také kdy je nutné v něm něco najít, například informaci, jak daná část funguje. Jelikož tyto scénáře popisují každodenní práci oddělení vývoje produktů, z toho vyplývá, že správně okomentovaný kód je nutnost. Je škoda, že autor tohoto kódu nevyužil ke komentování některý z populárních automatizovaných nástrojů k tomu určeným (např. Doxygen), který by ve výsledku vygeneroval přehlednou dokumentaci v PDF. Cyklus převzetí a kultivace jsem úspěšně zakončil inkrementací verze FW a vytvořením produkčního souboru pro automatické nahrávání FW pomocí nástroje Presto firmy ASIX.

Další fází bylo opravit některé nedostatky a chyby, které byly popsány v příslušném dokumentu. Začal jsem těmi nejlehčími a postupoval víceméně podobně jako v předchozí fázi. Zde zmíním především zkušenost s opravou jedné netriviální chyby.

Jednalo se o chybu, která byla zákazníkem definována takto: „Občas to přestane fungovat.“ Bohužel lepší definice nebyla k dispozici, avšak po komunikaci se zákazníkem se zjistilo, více: „Občas to přestane fungovat, ale pokaždé jiným způsobem. Stává se to zhruba jednou za týden až měsíc.“ V prvé řadě bylo nutné najít, kde je problém. Musím poznamenat, že jsem nebyl prvním, kdo se snažil tento problém vyřešit, proto jsem tušil, že se nebude jednat o banalitu. Provedl jsem různé testy, kdy mým cílem bylo dovést zdroj do nefunkčního stavu (bliká červená indikační LED, nereaguje na příkazy), leč neúspěšně. Po konzultaci s Ing. Navrátilem jsem byl pověřen vytvořením zátěžového testeru, který by odhalil příčinu. Tester jsem napsal v LabVIEW, pojmenoval „Stress Test“ a spolu s dalšími testery je budu detailněji popisovat v kapitole 2.4.1.

„Stress Test“ fungoval velice jednoduše, a to tak, že cyklicky prováděl definované testy (test základní komunikace, test proudového výstupu, test náhodnými znaky) na zdroji a zaznamenával uplynulý čas a chybovost. „Stress Test“ jsem zapnul v 9:00 a v 15:00 jsem měl první výsledky. Po analýze *logu* jsem zjistil, že zhruba po 5 hodinách intenzivní komunikace (cca 600 000 příkazů) dojde k jedné chybě. Jednalo se o chybu v komunikaci. Zpráva určená pro zdroj přišla z PC do MCU vždy rozdělena na dvě zprávy (PŘÍKAZ přišel jako P a ŘÍKAZ). Po několika opakováních pokusu a následné debatě s Ing. Navrátilem jsme se shodli, že se jedná o zákazníkem popisovaný problém. Zákazník proudový zdroj ovládá předem definovanou sekvencí příkazů v automatickém nadřazeném systému. Pokud jeden validní příkaz přijde jako dva nevalidní, dojde k selhaní celé sekvence (později se zjistilo, že existovala kombinace příkazů, které se po rozdělení na dva chovaly jako validní příkazy a zapříčinily pád systému, toto chování bylo vzápětí opraveno).

Vznikla definice problému: 1 z 600 000 příkazů přijme MCU zdroje jako dva, vždy nejprve první písmeno a pak zbytek. S touto informací šlo naložit o poznání lépe než s komentářem zákazníka. Zaměřil jsem se proto na komunikační modul. Během týdne jsem vyzkoušel mnoho věcí, například korekci hodnot registrů zodpovědných za taktování komunikačního rozhraní nebo zredukování akcí prováděných v ISR – obsluze přerušení přijímacího modulu, ovšem bez úspěchu. Celou dobu jsem měl ale tušení, že tento problém souvisí se synchronizací sdílených proměnných. CPU dané architektury má vlastnost, kdy před vstupem do ISR automaticky vypne přerušení a po výstupu je opět zapne. Proto se v mnoha případech u této

architektury synchronizace moc neřeší, nesmí se však zapomenout, že tento mechanismus funguje pouze v ISR a že synchronizace sdílených proměnných mimo ISR je nutné ošetřit manuálně.

S těmito vědomostmi jsem se zkusil na kód podívat z jiného úhlu, z pohledu strojových instrukcí, a řešení jsem měl na dlani. Porovnávací podmínka v jazyce C, byť je na jednom řádku, odpovídá více strojovým instrukcím. Nejprve se musí hodnoty proměnných nahrát do registrů CPU a pak porovnat. Zde vzniká prostor pro přerušení programu. Představím příčinu řešeného problému na modelové situaci (viz Obr. 10):

```
#include <iostream>
#include <chrono>
#include <thread>
#include <mutex>

volatile uint8_t a = 0;     // sdilena promenna a
volatile uint8_t b = 0;     // sdilena promenna b
bool run = true;
void ISR();
std::mutex mtx;

int main()
{
    auto t1 = std::chrono::high_resolution_clock::now();
    auto tISR = std::thread(ISR);
    while (1)               // smycka reprezentujici hlavni kod programu
    {
        mtx.lock();
        mtx.unlock();       // pokud zakomentujeme a nize odkomentujeme, program nikdy neskonci
        if (a != b)         // pokud se a nerovna b ukonci se program - kriticka podminka
        {
            auto t2 = std::chrono::high_resolution_clock::now();
            auto duration = std::chrono::duration_cast<std::chrono::microseconds>(t2-t1).count();
            std::cout << duration << " microseconds\n";
            run = false;    // chceme ukoncit vlakno ISR
            tISR.join();    // pockame na ukonceni vlakna ISR
            return 0;
        }
        //mtx.unlock();     // pokud se odkomentuje, (a != b) je vždy FALSE a problém je ošetřen

        // zde je zbytek programu
        // ...
    }
}

void ISR()                  // vlakno simulujici obsluhu preruseni „ISR“ v MCU
{
    int c = 0;
    while (run)
    {
        mtx.lock();
        a ^= 0x1;           // XOR prvniho bitu promenych a, b
        b ^= 0x1;           // a, b se v „ISR“ zmeni obe „najednou“, (a != b) má být FALSE
        mtx.unlock();       // pokud se tak ale stane „uprostřed“ podmínky (a != b), je problém
        c++;
    }
    std::cout << c << " ISR iters\n";
}
```

Obr. 10: Ukázkový C++ program simulující diskutovanou chybu synchronizace ve FW

Je třeba porovnat sdílenou proměnnou **a** se sdílenou proměnnou **b** a v závislosti na výsledku provést akci. CPU nejprve nahraje hodnotu **a** do prvního registru. Náhle dojde k přerušení, kontext je uložen na zásobník a začne se vykonávat obsluha přerušení. Zde dojde ke změně hodnot proměnných **a** i **b**. Po ukončení dojde k obnově kontextu a CPU pokračuje nahráním změněné hodnoty proměnné **b** do svého druhého registru. Jelikož ale v prvním registru má stále předchozí nezměněnou hodnotu proměnné **a**, následující porovnání obou hodnot vyústí ve falešnou informaci.

Následky můžou i nemusí být kritické, záleží, jak se falešná informace projeví. V našem případě se projevila jako chyba komunikace v 1 z 600 000 případů. Tomu všemu lze předejít pouhými dvěma instrukcemi. Před podmínkou se přerušení vypne a po skončení se opět zapne. Následné několikadenní testování pomocí SW „Stress Test“ prokázalo, že chyba byla odstraněna, zákazníkovi se přehrál FW a problém byl vyřešen. Chyba ve FW proudového zdroje napájejícího LED moduly není až tak kritická v porovnání se slavnou chybou FW radioterapeutického počítače Therac-25 z 80. let, kde právě podobná chyba synchronizace přístupu ke sdíleným proměnným (*race condition*) zapříčinila smrt tří pacientů a vážné zranění dalších čtyř. [7]

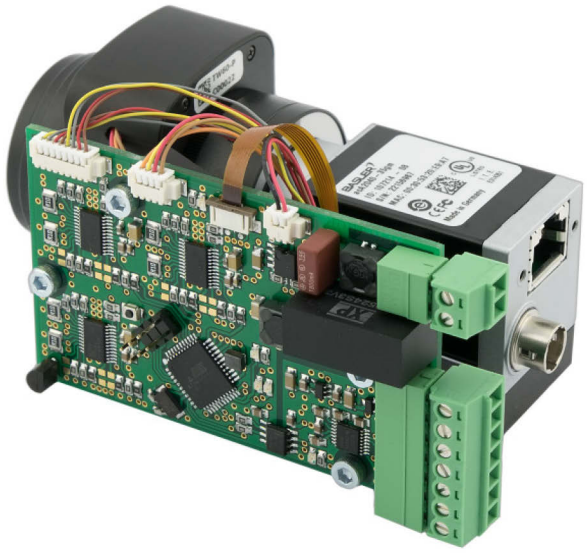
Poslední fází bylo přidání nových funkcí. Tento proces je dnes u mnoha produktů prováděn pravidelně v určitém intervalu, kdy se za nějakou dobu nashromáždí poznámky jak od zákazníků, tak ze strany vývojářů, které vedou k přidání nebo opravení funkčnosti. Tyto poznámky jsem zpracoval, zkonzultoval a následně implementoval. Šlo například o integrování automatického nahrávání FW pomocí specializovaného SW tzv. *bootloader*, zpřesnění měřících konstant pro zlepšení parametrů, přidání nových příkazů (např. restart systému, vyčtení rozsahů limitů, přepnutí rozsahu proudu, nastavení a vyčtení sériového čísla, revize a názvu zařízení) nebo zvýšení přesnosti výpisu čísel s desetinnou čárkou. Veškerá práce na FW byla úspěšná, potvrzuje to fakt, že aktuálně se v průmyslu nachází více než 150 kusů proudového zdroje, z toho většina obsahuje verzi FW, na které jsem pracoval.

## 2.2 Vývoj ovladačů v jazyce C++

### 2.2.1 Ovladač „Řídící jednotky pro objektivy s krokovými motory“

„Řídící jednotka pro objektivy s krokovými motory“, dále už jen „P-IRIS Controller“, je produkt firmy ATEsystem (viz Obr. 11), určený pro použití v dopravních systémech, jako je třeba sledování průjezdu křižovatkou nebo úsekové měření rychlosti. Jeho hlavní funkcí je propojit motorizovaný objektiv s krokovými motory (*P-Iris*) s kamerou Basler ace, a poskytnout uživateli jejich vzdálenou správu pomocí PC, nebo jiného řídícího systému. Napájí se stejnosměrným napětím 12–24 V a prodává se buď ve verzi RS232, kdy je třeba do nadřazeného systému vést dva kabely (jeden sériový RS232 pro „P-IRIS Controller“ a druhý UTP pro zapojení kamery do sítě LAN) anebo ve verzi UART, kdy stačí vést jediný kabel (UTP pro kameru). Na zakázku upravený firmware kamery Basler zařídí UART komunikaci s jednotkou lokálně pomocí kamerových digitálních I/O.

Klíčové funkce jsou ovládání clony, ostření a přiblížení, dále je zde možnost přepínat IR filtr, což umožňuje používat kameru i v noci. Ovládání probíhá pomocí textového komunikačního protokolu standardně rychlostí 9600 baudů/sekundu, ve verzi UART pak rychlostí 4800 baudů/sekundu, z důvodu emulování signálu na digitálních vstupech/výstupech kamery, a tudíž zhoršených přenosových vlastnostech. Nutno podotknout, že pro danou aplikaci obě rychlosti plně dostačují. Textové příkazy jsou zakončovány standardně pomocí znaků 0x0D, 0x0A (CR, LF) a kompletní specifikaci protokolu lze nalézt v katalogovém listu produktu. [8]



Obr. 11: „Řídící jednotka pro objektivy s krokovými motory“ – pohled na PCB

Stávající softwarová podpora pro „P-IRIS Controller“ zahrnovala LabVIEW obslužnou aplikaci a LabVIEW ovladač. Byly zde však dva problémy. První problém byl, že LabVIEW obslužná aplikaci podporuje pouze RS232 verzi produktu. To z toho důvodu, že rozšíření o UART podporu by znamenalo nutnost závislosti na nestandardních modulech. Ve výsledku by se to

projevilo jako několikanásobné zvětšení velikosti distribuované aplikaci, nutnost přítomnosti onoho modulu na všech koncových PC (jedná se o modul NI Vision Development Module) a v neposlední řadě také zvýšení ceny, protože tento modul je komerční produkt. Druhý problém byl, že LabVIEW není ideální platforma pro venkovní dopravní aplikace, znamená totiž závislost na PC s OS Windows (podpora OS Linux je tristní) s nainstalovaným SW „LabVIEW Run-Time“. Tato konfigurace ale není vždy dostupná, mnoho výrobců stále častěji spoléhá na levné a spolehlivé řešení nad ARM ekosystémem, kde dominují unixové OS či RTOS. Mým úkolem bylo vyřešit tyto dva problémy během dvou měsíců.

Nejprve jsem zanalyzoval celou situaci. Měl jsem za úkol zprovoznit „P-IRIS Controller“ jak v RS232, tak UART/Ethernet verzi za použití co nejmenšího množství SW závislostí a s důrazem kladeným na univerzálnost, výkonnost a kompatibilitu s různými systémy (v první řadě OS Windows, v druhé řadě OS Linux). Jako jazyk jsem zvolil C++17. Prvním mezikrokem vývoje měla být dynamická, resp. statická knihovna pro Windows (DLL, LIB) a pro Linux (SO, A). Dále se mělo pokračovat integrováním ovladače do *frameworku* UMDF. Začal jsem návrhem architektury. Jako centrální bod ovladače jsem zvolil třídu *Factory*, která dle návrhového vzoru *Factory* a *Singleton* spravuje instance pro konkrétní komunikační rozhraní – RS232 a UART/Ethernet. Díku tomu je možné paralelně ovládat více produktů „P-IRIS Controller“ jediným unikátním identifikátorem. Veřejné metody třídy *Factory* jsou:

```
DevID_t CreateDevice(Mode mode = Mode::ETHERNET);
DevID_t CreateDevice(const std::string& name, Mode mode);
IDevice* GetDeviceInstance(DevID_t id);
std::string GetDeviceName(DevID_t id);
Status RemoveDevice(DevID_t id);
```

Obr. 12: „P-IRIS Controller“ – veřejné metody třídy *Factory*

Hlavní a jediné rozhraní vystavené uživateli se jmenuje *IPiris*. Je to ve skutečnosti *smart pointer* typu *IDevice* (viz Obr. 13), což zaručuje moderní minimalizaci úniků paměti díky vlastnostem jazyka (skončení doby života objektu na konci platnosti lokální proměnné). [9] Použití *smart pointerů* je dnes u jazyků bez GC nutností, protože nesprávně implementovaná správa paměti nebo špatně použitá práce s pamětí může vždy vést k únikům paměti a v konečném čase tak třeba k pádu systému. Nebo hůře, ke kompromitaci systému. Jeden z nejznámějších úniků paměti v populárním FOSS OpenSSL (kryptografická knihovna používaná např. v implementacích TLS – HTTPS) šel zneužít útočníky k získání privátních šifrovacích klíčů vzdáleného systému a tím jeho kompromitaci, dostal jméno *Heartbleed*. [10]

```
typedef SmartPointer<IDevice> IPiris;
```

Obr. 13: „P-IRIS Controller“ – veřejné rozhraní je *smart pointer* typu *IDevice*

Toto rozhraní bylo napojeno na veřejné exportované funkce knihovny. Jako volající konvenci jsem zvolil CDECL (argumenty funkce jsou vloženy do zásobníku v opačném pořadí a volající jej maže) z důvodu kompatibiliy s OS Linux oproti STDCALL (volaný maže zásobník), kterou využívá především OS Windows. Zatím vypadlo vše v naprostém pořádku, níže (viz Obr. 14) lze vidět některé hlavní veřejné exportované funkce a jejich argumenty.

```
extern "C" EXPORT DeviceID_t CDECL Open(DeviceName_t name, Mode mode);
extern "C" EXPORT Status_t CDECL Close(DeviceID_t id);
extern "C" EXPORT Status_t CDECL Search(Mode mode, char found[10][100]);
```

Obr. 14: „P-IRIS Controller“ – koncept některých veřejně exportovaných funkcí

Pokračoval jsem fází implementace, kdy během jedné porady přišel zlomový okamžik. Ovladač v režimu UART/Ethernet využívá aktivní relaci kamery k obsluze „P-IRIS Controlleru“ pomocí emulovaného rozhraní UART. Tato relace je unikátní a lze s každou kamerou navázat maximálně jednu, jedná se o TCP spojení. Z jedné strany to jako problém nevypadalo, ovladač se kameře připojí a následně s jednotkou úspěšně komunikuje, nicméně problém nastal, když jsem se chtěl podívat na obraz z kamery. Aktivní spojení ovladače s kamerou za účelem řízení objektivu zabraňovalo navázání spojení s kamerou za účelem práce s obrazem. Tento na první pohled jasný fakt zůstal všem až doposud skryt. RS232 verze tento problém samozřejmě neměla, protože ovladač komunikoval s jednotkou po sériové lince a obraz z kamery uživatel zpracovával po Ethernetu. UART/Ethernet verze však kombinuje tyto dva kanály do jedné linky a tato přehlédnutá chyba v koncepci mě vrátila zpět, do fáze návrhu architektury.

Znamenalo to pro mě to, že musím zajistit, aby měl uživatel možnost kdykoliv posílat libovolné zprávy objektu kamery. Posílat libovolné zprávy přes univerzální ABI dynamické knihovny samozřejmě nelze, musel jsem tedy jít jinou cestou. Jenda z možností byla nechat vytváření objektu kamery na volajícím (uživateli). Posílání ukazatelů na objekty přes C++ ABI nemá s univerzálností a kompatibilitou nic společného, zbývala tak už snad jediná možnost, a to registrace *callback* funkce (resp. funkcí pro čtení a zápis bajtů přes rozhraní kamery). Toto ne příliš intuitivní řešení bylo zamítnuto a po konzultaci s Ing. Navrátilem jsem změnil celou koncepci. Ovladač jako binární distribuce (dynamická knihovna) byl nahrazen ovladačem jako zdrojový kód. Zákazník/uživatel ho ke své aplikaci jednoduše připojí a veškeré zmíněné problémy jsou vyřešeny, za cenu mírného snížení komfortu (s každou novou verzí ovladače musí být překompilována i zákaznická aplikace).

Změna architektury a následná implementace probíhala bez problémů, třída *Factory* zůstala téměř beze změn, veřejné exportované funkce byly odstraněny, hlavním veřejným rozhraním ovladače se stala místo funkcí (viz Obr. 14) přímo třída *IDevice (*viz Obr. 15*)*.

Metody v prvním odstavci slouží k otevírání a zavírání spojení s jednotkou, třída *PirisDevice* slouží jako kontejner definující typ zařízení, resp. ukazatel na něj. Druhý odstavec obsahuje metody pro vyčítání parametrů z jednotky jako název, verze FW, typ objektivu, maximální poloha, aktuální poloha a stav, v třetím odstavci je řešen reset a inicializace a metody v posledním odstavci pak slouží pro nastavování parametru objektivu (ostření, clona, přiblížení a IR filtr). Datový typ *tuple* (C++11) jsem zvolil z toho důvodu, aby se veškeré hodnoty vracely přes hodnotu (na zásobník), čímž se minimalizují možné úniky paměti a obecně zrychlí program (zásobník se na rozdíl od haldy může vejít do *L3*, resp. *L2 cache*).

```
Interface IDevice
{
public:

    virtual Status Open(const PirisDevice& dev, VerboseLevel verbose = VerboseLevel::NONE) = 0;
    virtual Status Close() = 0;

    virtual std::tuple<StatusEx, DataID> ReadID() = 0;
    virtual std::tuple<StatusEx, DataPosition> ReadPosition() = 0;
    virtual std::tuple<StatusEx, DataParams> ReadParams() = 0;
    virtual std::tuple<StatusEx, DataState> ReadState() = 0;

    virtual StatusEx DevReset() = 0;
    virtual StatusEx DevHoming() = 0;

    virtual StatusEx SetAbsolute(const FocusZoomIris<uint16_t>& values, bool ir_filter = false) = 0;
    virtual StatusEx SetRelative(const FocusZoomIris<int16_t>& values) = 0;
};
```

Obr. 15: „P-IRIS Controller“ – část konečné implementace veřejného rozhraní

Pro vyhledávání jednotek slouží dvě statické metody, vracející vektor příslušného objektu přes referenci (viz Obr. 16). Metody jsem zvolil jako statické, protože si neuchovávají žádný stav a můžou být volány kdykoli a kýmkoli, přičemž nehrozí riziko souběhu.

```
static int16_t ScanEthernet(std::vector<Pylon::CDeviceInfo>& devices, bool verbose = false);
static int16_t ScanSerial(std::vector<serial::PortInfo>& devices, bool verbose = false);
```

Obr. 16: „P-IRIS Controller“ – jediné dvě statické bez stavové metody pro vyhledávání

Ke kódu ovladače jsem následně napsal demo s příklady užití – spustitelný soubor, demonstrující veškeré funkcionality pomocí systémové příkazové řádky (viz Obr. 18). Demo i ovladač jsou multiplatformní, úspěšně jsem je testoval na OS Windows 7, 8, 10 (amd64, x86) a OS Linux Ubuntu 18 (amd64, x86, armhf a aarch64), jedná se tedy o minimálně 6 podporovaných platforem. Jelikož standardní knihovna jazyka C++ nenabízí multiplatformní podporu pro sériovou komunikaci (RS232), použil jsem populární FOSS knihovnu od Williama Woodalla a Johna Harrisona. Pro komunikaci s kamerou Basler jsem použil binární podporu přímo od

firmy Basler a to Pylon 5 spolu s generickým kamerovým rozhraním GenICam 3. Tyto knihovny tak činí jediné závislosti ovladače, v Linuxu se jedná zhruba o 10 MB dynamických knihoven. Ve Windows jde o zhruba 5 MB, je však třeba mít nainstalovaný „VC++ Runtime“ 2013 a 2017 o velikosti zhruba 10 MB. Vzhledem k původním velikostem LabVIEW distribuce se ale jedná o zanedbatelné položky. Výsledné spustitelné demo integrující ovladač má v OS Windows velikost zhruba 100 KB, v OS Linux zhruba 2 MB. Níže (viz Obr. 17) je nastíněn typický příklad použití, ve kterém se nejprve vytvoří objekt ovladače, připojí se k jednotce, vyčte se ID a pozice, nastaví se dané hodnoty, objektiv se resetuje a dojde k odpojení.

```
auto dev_id = ATEsystem_PIRIS::Factory::getInstance()->CreateDevice(ATEsystem_PIRIS::Mode::ETHERNET);
auto piris  = ATEsystem_PIRIS::Factory::getInstance()->GetDeviceInstance(dev_id);
piris->Open((ATEsystem_PIRIS::PirisDevice)(devices_serial[dev_chosen]));

auto[status_ex1, data_id] = piris->ReadID();        // read device name and firmware version
auto[status_ex2, data_p1] = piris->ReadPosition();  // read actual position of motors
piris->SetAbsolute(20, 40, 10, true);               // set focus = 20, zoom = 40, iris = 10, ir = on

piris->DevReset();
piris->Close();
```

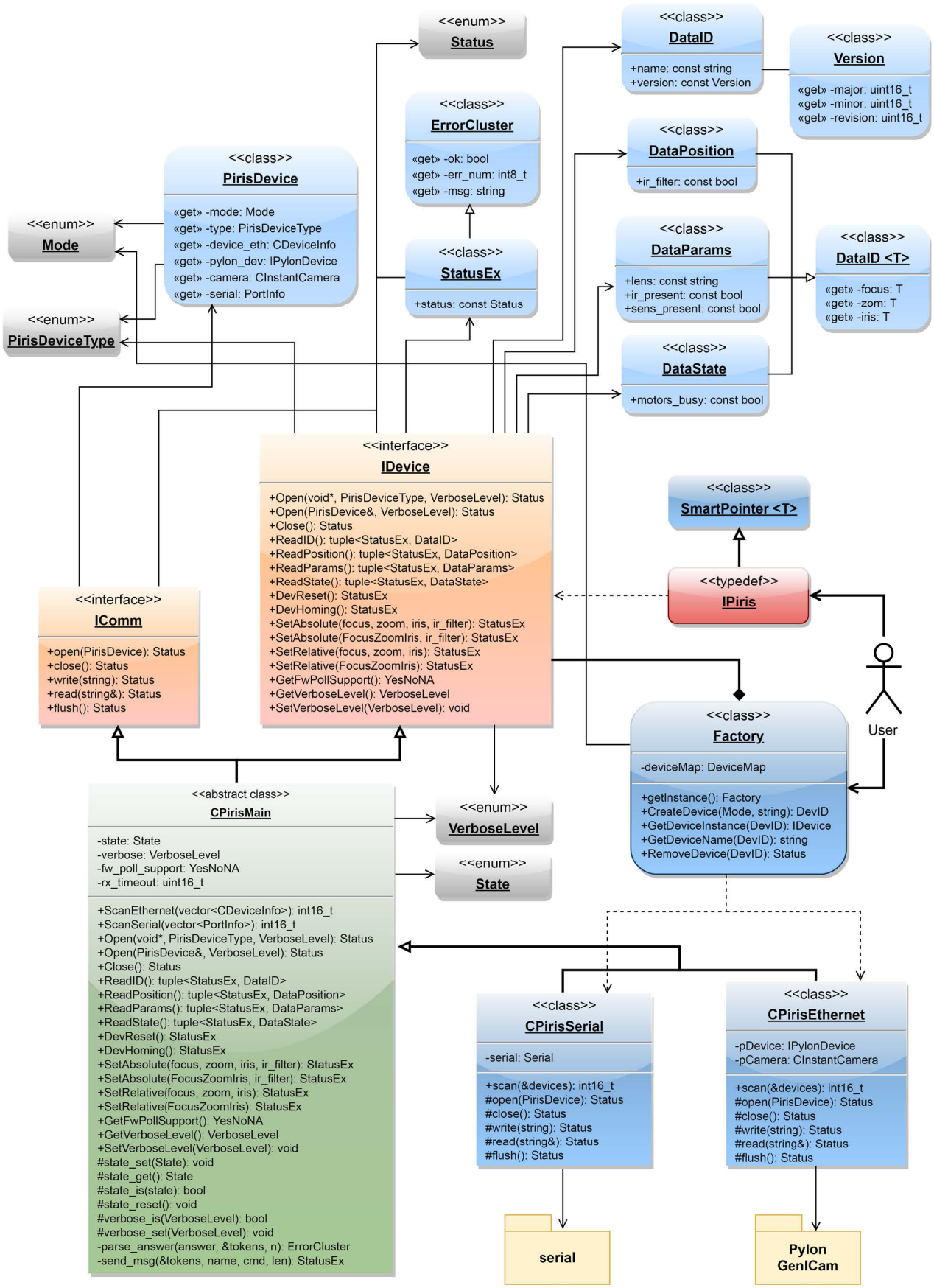
Obr. 17: „P-IRIS Controller“ – příklad použití ovladače

S výsledkem jsem já i můj nadřízený spokojen, splnil jsem veškeré požadavky a myslím, že jsem udělal i něco navíc. Ovladač pro „P-IRIS Controller“ je nyní multiplatformní s minimem závislostí, připraven na použití v libovolné aplikaci. Trošku mě mrzí, že nešlo jít cestou UMDF a dynamické knihovny, protože pak by šlo integrovat ovladač téměř s jakýmkoli jazykem a prostředím (LabVIEW, C#), vlastnost architektury mi v tom ale zabránila. Ovladač firma uvolnila jako FOSS, to znamená že včetně zdrojových kódů, spustitelných souborů, konfigurací IDE i dokumentace (viz Obr. 19) ho lze stáhnout z firemní webové stránky. [11]

```
jakub@ubuntu:~/ $ ./ATEsystem.PIRIS-driver-Linux-ARM64.out
This is ATEsystem P-IRIS Controller driver example.
www.atesystem.cz  produkty@atesystem.cz
================================================

Choose device connection mode:
  0 : Serial
  1 : Ethernet
> 1


Choose demo mode:
  0 : User input
  1 : Sequence
> 0

Choose device number:
  0. Basler acA2040-35gmATE#003053267EC2#192.168.15.140:3956
> 0

PIRIS open - success (Ethernet camera (camera overiden))
Camera Device Information
=========================
Vendor           : Basler
Model            : acA2040-35gmATE
Firmware version : 107603-01;U;acA2040_35gmATE;V1.0-2;0
=========================
PIRIS set up - success
```

```
.\ATEsystem.PIRIS-driver-WinNT-x64.exe

This is ATEsystem P-IRIS Controller driver example.
www.atesystem.cz  produkty@atesystem.cz
================================================

Choose device connection mode:
  0 : Serial
  1 : Ethernet
> 1


Choose demo mode:
  0 : User input
  1 : Sequence
> 0

Choose device number:
  0. Basler acA2040-35gmATE#003053267EC2#192.168.15.140:3956
> 0

PIRIS open - success (Ethernet camera (camera overiden))
Camera Device Information
=========================
Vendor           : Basler
Model            : acA2040-35gmATE
Firmware version : 107603-01;U;acA2040_35gmATE;V1.0-2;0
=========================
PIRIS set up - success
```

Obr. 18: „P-IRIS Controller“ – demo v OS Linux (vlevo) a v OS Windows (vpravo)



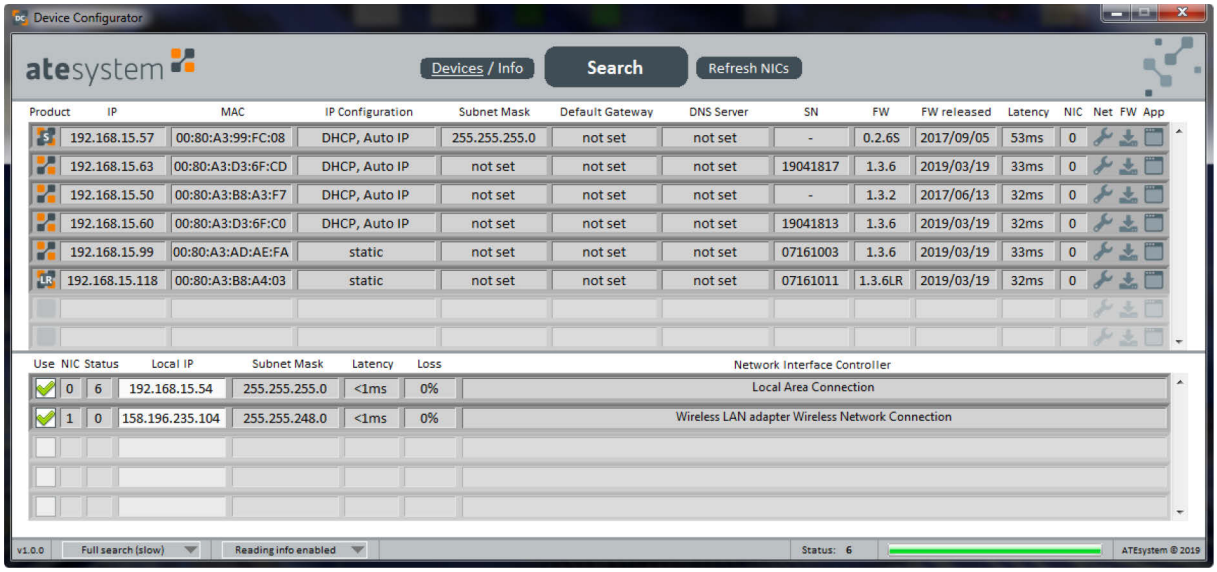

Obr. 19: „P-IRIS Controller“ – třídní diagram v UML

## 2.3 Vývoj aplikací v prostředí LabVIEW

### 2.3.1 Nástroj pro správu „Proudových zdrojů pro LED moduly“

„Proudový zdroj pro LED moduly“ byl představen v kapitole 2.1.1. Jeho SW podpora představovala především LabVIEW ovladač a LabVIEW obslužnou aplikaci, která pomocí ovladače sloužila k základní manipulaci se zdrojem. Po spuštění aplikace bylo nutné nejprve zadat IP adresu zdroje, následně šlo nastavovat a vyčítat elektrické veličiny a další parametry. Během vývoje FW jsem tuto aplikaci intenzivně využíval k testování, postrádal jsem však možnost automatického vyplnění IP adresy aktuálně připojeného zdroje k síti. Proces tedy vypadal tak, že před zapnutím aplikace jsem musel prvně vždy spustit oficiální aplikaci od výrobce Ethernet/UART převodníku (integrovaného na PCB zdroje), vyhledat dostupná zařízení, IP adresu zkopírovat a následně vložit do firemní ovládací aplikace. Zdálo se mi to zbytečně komplikované, proto ve volné chvíli, kdy zrovna probíhalo dlouhodobější zátěžové testování, jsem pomocí síťového analyzátoru Wireshark odposlechl protokol vyhledávací aplikace a během jednoho dne napsal jednoduchý program v LabVIEW, který fungoval velice podobně. Neměl jsem jinou možnost, protože na internetu žádná alternativa neexistovala. Kolem programu jsem vytvořil jednoduché GUI a nahradil jím tak oficiální aplikaci. Během dalších testů a vývoje FW nyní šlo intuitivně vyhledat proudové zdroje v síti LAN jedním kliknutím tlačítka.

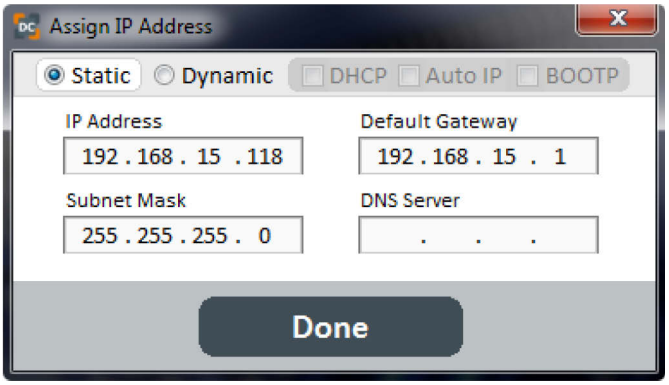
Díky jednoduchosti a praktičnosti se tento program začal v oddělení vývoje používat a mým úkolem během následujícího 1 měsíce bylo pokračovat na vývoji GUI aplikace, tehdy pojmenované „ATEpwr inspector“, integrující zmíněný vyhledávací algoritmus (viz Obr. 6). Šlo především o to přidat další funkcionalitu (zákaznické požadavky), zapracovat na reprezentativním vzhledu, vylepšit vyhledávácí schopnosti, otestovat a vytvořit tak stabilní aplikaci včetně instalátoru, která by šla distribuovat zákazníkovi. Výsledek lze vidět níže (viz Obr. 20):



Obr. 20: „Device Configurator“ – čelní panel hlavního okna aplikace

Nejprve jsem převedl GUI aplikace do firemní šablony. Pak jsem pokračoval prací na hlavní funkcionalitě, která musela být spolehlivá a robustní, a sice vyhledávání zdrojů v síti LAN. Vyhledávání funguje pomocí paketů UDP *broadcast*, kdy na specifickou sekvenci bytů odpoví Ethernet/UART převodník Lantronix identifikační zprávou, opět v předem definovaném formátu. Tento mechanismus funguje spolehlivě v rámci stejné podsítě i v případě, kdy je na koncovém zařízení nastavena jiná podsíť, ale mezi zařízeními není žádný směrovač. Nicméně v závislosti na konfiguraci směrovače pak může být UDP paket směrován i do jiné sítě. Toho lze využít v případě, kdy uživatel nastavil IP adresu mimo rozsah aktuální podsítě a pomocí TCP se tak ke zdroji nepřipojí. I když není v cestě žádný směrovač, při pokusu o navázání TCP spojení vyhodnotí lokální TCP/IP *stack* IP jako adresu mimo rozsah aktuální podsítě a odešle uživateli ICMP zprávu *Destination Unreachable* (požadavek zahodí). Jednou z cest pak je technika UDP *broadcast*.

Podobně jako vyhledávání, funguje i nastavování síťových parametrů. Opět pomocí UDP lze správně naformátovanou zprávou (v binární formě) nastavit IP adresu, masku podsítě, DNS server a výchozí bránu, kdy klíčem k identifikaci zařízení je MAC adresa. Lze si samozřejmě vybrat mezi statickou IP adresou a dynamickou. Technologie, které jdou nastavit v rámci dynamické IP adresy jsou: BOOTP (pro zpětnou kompatibilitu se staršími systémy), DHCP (dnes standard) a Auto IP (automatické přiřazení tzv. *Link-local* IP adresy v nepřítomnosti směrovače). Nastavování síťových parametrů je tedy druhou hlavní funkcionalitou hned po vyhledávání a lze ji vidět níže (viz Obr. 21):

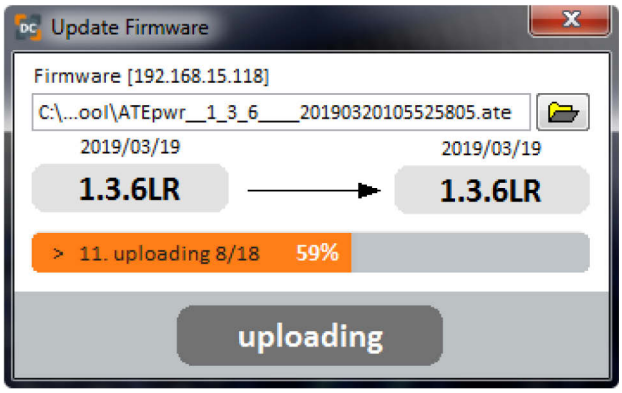



Obr. 21: „Device Configurator“ – čelní panel okna nastavování síťových parametrů

Dále jsem přidal možnost skupinového vyčítání hlavních parametrů zdroje a elektrických veličin. Toto je užitečné při vzdálené inspekci více proudových zdrojů např. v rozváděči, zákazník ihned vidí stavové indikátory přehledně pod sebou, bohužel tato funkcionalita je dostupná pouze když proudový zdroj nemá aktuálně otevřené žádné TCP spojení. Do dokumentu obsahujícího podněty pro případné budoucí vylepšení produktu jsem proto zanesl poznámku zkusit vyřešit tento problém pomocí UDP.

Další významná schopnost je vzdáleně přehrát FW proudového zdroje. Vzdálený update zajišťuje *bootloader* a obslužný SW napsaný v OOP LabVIEW, který jsem integroval do této aplikace (viz Obr. 22). Jde také o mou práci, byla však již do detailů popsána v samostatném předmětu Semestrální Projekt, proto by bylo nevhodné ji zde diskutovat, ač se jedná o zajímavou problematiku. Poslední významnou funkcionalitou je integrace ovládací aplikace, která lze spustit ke každému nalezenému zdroji zvlášť jedním kliknutím tlačítka, kdy dojde k automatickému připojení a uživatel může ihned používat veškeré funkce zdroje.

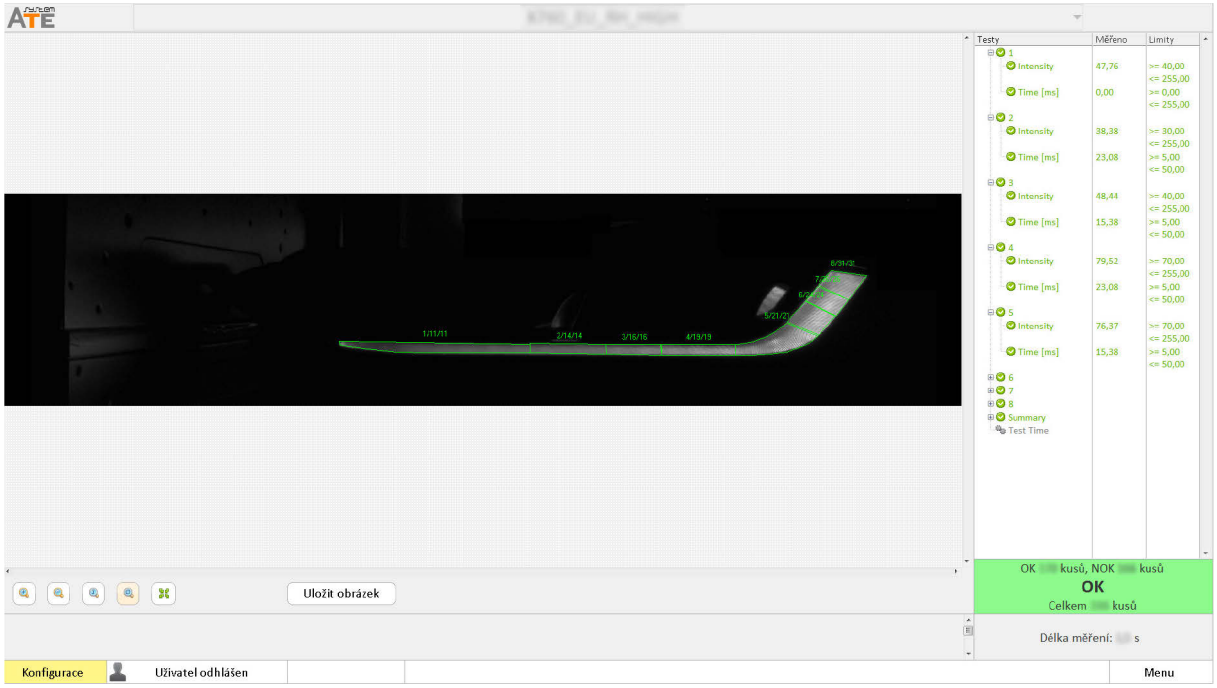
Tento SW dostal jméno „Device Configurator“, vytvořil jsem k němu instalátor i manuál a vše je dostupné na firemní webové stránce. [12] Úkol dopadl úspěšně, „Device Configurator“ zcela integroval, a tedy i nahradil původní obslužnou aplikaci, vytlačil používání vyhledávacích aplikací třetí strany a poskytl zákazníkům i vývojářům komfort jednotného víceúčelového SW pro správu produktu „Proudový zdroj pro LED moduly“ firmy ATEsystem.




Obr. 22: „Device Configurator“ – čelní panel okna vzdálené aktualizace FW

### 2.3.2 Tester animovaného blinkru světlometů osobních automobilů

Mým dalším úkolem bylo vymyslet a naimplementovat algoritmus pro analýzu animovaného blinkru moderních světlometů osobních automobilů. Firma ATEsystem se již delší dobou zabývá testováním různých typů světlometů v průmyslovém prostředí, animovaný blinkr je však něco nového, co teprve přichází na zdejší trh. Za poslední dobu lze vypozorovat znatelný nárůst poptávky po animovaném blinkru jak ze strany spotřebitelů, tak ze strany výrobců, kteří potřebují světlomet s animovaným blinkrem řádně otestovat před distribucí zákazníkovi. Tento trend bude pravděpodobně v budoucnu pokračovat, byť je to pouze má spekulace. Historie firmy ATEsystem tak logicky potvrzuje její zájem o trh s animovanými blinkry. Během 2 týdnů jsem měl za úkol přispět do firemního *know-how* mým algoritmem. Výsledek mé práce jako součást systému pro vizuální inspekci lze vidět níže (viz Obr. 23), kde právě úspěšně proběhlo testování světlometu s animovaným blinkrem tvořeným 8 segmenty.




Obr. 23: Čelní panel systému vizuální inspekce včetně *plug-inu* pro analýzu blinkru

Jelikož se jedná o aktuálně využívaný SW, který je součástí výrobního řetězce prémiových světlometů, nemohu zmínit detaily návrhu architektury ani jiné podrobnější informace, zaměřím se proto na průběh vývoje. Nejprve jsem potřeboval data, abych mohl začít přemýšlet nad návrhem algoritmu. K dispozici mi byl před produkční vzorek světlometu osobního automobilu. Za pomocí řídící jednotky připojené ke sběrnicovému konektoru světlometu šlo digitálním signálem ovládat animaci blinkru. Dále mi byla k dispozici kamera Basler s rozhraním Ethernet, která měla stejné parametry, jako použitá kamera v systému vizuální inspekce, jenž měl být rozšířen mým *plug-inem* o schopnost analýzy animovaného blinkru. Kameru jsem připojil

k PC a pomocí SW Pylon nasnímal obrázky sekvence animace rychlostí 120 FPS na disk (rychlost v průmyslovém prostředí pomalá, avšak pro účely této aplikace plně postačující). Po zběžném prohlédnutí jsem provedl mírné korekce v nastavení (clona, doba expozice, ostření, úhel natočení, osvětlení) a následně jsem započal práci na algoritmu.

Animace byla u konkrétní produktové řady řešena LED diodami, seskupených do jednotlivých segmentů, ovládanými digitálně pomocí vestavěné elektroniky. Zadání znělo: rozlišit vadný kus od bezchybného. Dále jsem věděl, že obsluha *plug-inu* bude znát minimální a maximální čas délky svícení každého segmentu. Poslední informací byl maximální čas od začátku animace do úplného rozsvícení blinkru, který byl stanoven z vyšších pozic zadavatele projektu na 200 ms. I když se to na první pohled nemusí zdát, korektní analýza tohoto problému je relativně komplexní. Jak jsem již v úvodu zmínil, nemusel jsem řešit problémy týkající se barvy světla a intenzity LED diod včetně dalších charakteristik spojených s vlastnostmi světla, jelikož tyto parametry nesouvisí s animací a analyzují se v jiné fázi testovací sekvence. Problém tedy mohl nastat s jedinou LED diodou, s celým segmentem, s řídící elektronikou, se špatně vyrobeným difuzorem, který se nachází před segmenty, nebo s libovolnou kombinací těchto faktorů. Tento problém se pak mohl projevit chybou v animaci, která mohla například konkrétně vypadat jako nerozsvícení jednoho segmentu, rozsvícení segmentu ve špatném pořadí, zhasnutí už rozsvíceného segmentu v nesprávný čas, překročení délky maximální doby rozsvícení celé animace nebo vadný či chybějící FW řídící elektroniky (rozsvícení všech LED diod najednou místo animace).

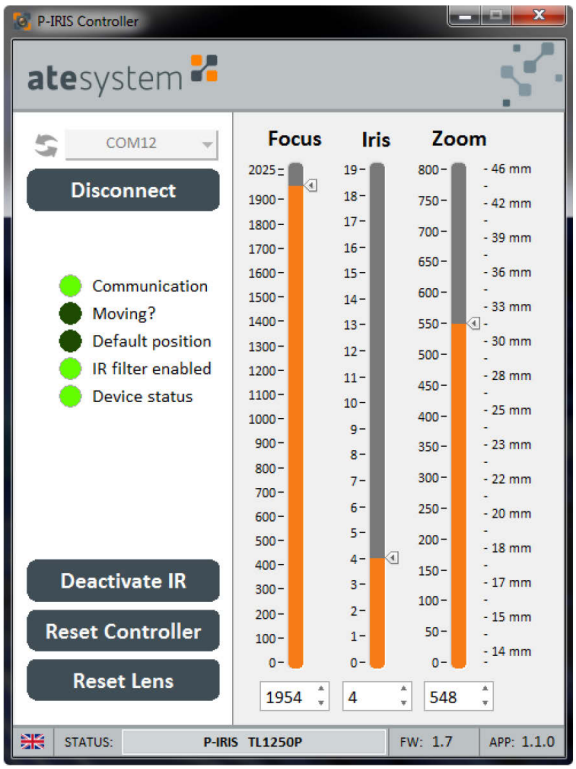
Během práce jsem narazil jednu překážku, která zkomplikovala návrh i implementaci, a sice difuzor. Díky přítomnosti difuzoru před LED diodami, který zajistí nezbytné rozptýlení světla, a tedy i plynulost přechodu mezi jednotlivými segmenty, docházelo ke značnému kolísání uniformity snímaných dat ze stejné animace stejného kusu v neměnném prostředí. Pro analýzu animace blinkru by bylo mnohem spolehlivější snímat kamerou namířenou přímo na LED diody, nicméně zákazník měl požadavky, které znamenaly testování až po montáži vnější mechanické konstrukce včetně difuzoru, takže jsem se tomu musel přizpůsobit. Nakonec se místo použití filtrů ukázalo vhodnější lehce zvýšit toleranci některých časových parametrů, ideální by však bylo snímat obraz z blinkru bez difuzoru.

Úkol jsem nakonec úspěšně zvládl, *plug-in* jsem integroval a nasadil do systému vizuální inspekce u zákazníka, kde už nyní slouží jako jeden z mnoha faktorů výstupní kontroly světlometů osobních automobilů. Implementovat v LabVIEW analýzu obrazu pomocí knihoven NI-IMAQdx je přímočaré, spolehlivé, a především časově nenáročné, proto bych příště pro úlohu stejného charakteru zvolil opět jazyk G a prostředí LabVIEW.

### 2.3.3 Obslužná aplikace „Řídící jednotky pro objektivy s krokovými motory“

Produkt „P-IRIS Controller“ byl představen v kapitole 2.2.1, nyní se zběžně zaměřím na obslužnou aplikaci pro PC. Mým úkolem bylo během 1 týdne dokončit rozdělanou aplikaci, která měla poskytnout uživatelům produktu „P-IRIS Controller“ jednoduché a intuitivní ovládání, především během rychlých testů nebo převáděcích akcí. Na rozdíl od ovladače, který je cílen na reálný provoz a který nabízí kompletní API, tato aplikace má sloužit jako jeho doplněk, s funkčností omezenou jejím potřebám. Aplikace byla napsaná v LabVIEW, s ohledem na firemní prostředí a *know-how* to dávalo smysl, byť se později ukázalo, že se jednalo o nešťastnou volbu (díky některým faktorům podporuje pouze verzi RS232, více v kapitole 2.2.1).

Nejprve jsem provedl analýzu kódu. Jednalo se o populární návrhový vzor QMH, který je použit i v dalších v této práci diskutovaných LabVIEW aplikacích. [13] Bohužel jsem objevil několik návrhových chyb, které odstranit by vyžadovalo více času, než jsem měl k dispozici. Nicméně opravit (zalepit) je šlo relativně jednoduše a na funkčnosti se to nijak neprojevilo. Podobné chyby v koncipování jádra SW většinou nejdou lehce odstranit, jejich oprava je možná často za cenu zhoršení přehlednosti a čistoty kódu, což vede v budoucnu ke zvýšení nákladů na údržbu či rozšiřování. Proto je někdy lepší špatně navržený SW zahodit a začít od začátku, kdy analýzu takové situace a finální rozhodnutí nejčastěji provádí vedoucí pracovník produktového oddělení.




Obr. 24: Čelní panel obslužné aplikace pro „P-IRIS Controller“

Po opravách, které vedly ke stabilně fungující verzi aplikace, jsem pokračoval rutinními záležitostmi týkající se sjednocení grafických prvků do stylu firemní šablony, vytvoření druhé jazykové mutace, schopnosti vyhledat dostupná připojená zařízení a podpory nového FW, tedy i nových funkcí. Spolu s prací na kódu jsem samozřejmě vytvářel příslušnou dokumentaci, vždy se jedná minimálně o soubor popisující verze a změny v nich provedené (tzv. *changelog*) a manuál, dokument shrnující vše potřebné k používání daného SW. Úkol jsem dokončil úspěšně, SW nyní funguje spolehlivě jako obslužná aplikace pro produkt „P-IRIS Controller“, kdy uživatel po volbě příslušného zařízení může nastavovat a vyčítat parametry objektivu a provést inicializaci nebo restart jednotky (viz Obr. 24). Jedná se o podmnožinu funkcionality ovladače v jazyce C++. Vytvořil jsem spustitelný soubor a instalátor pro OS Windows, který spolu s dokumentací lze stáhnout z firemní webové stránky.

### 2.3.4 Benchmark grafických a video formátů

Posledním úkolem z kategorie LabVIEW aplikací diskutovaných v této práci bylo vytvoření *benchmarku* grafických a video formátů. Jinými slovy jsem dostal za úkol zjistit, který *kodek* by měli použít kolegové z oddělení systémové integrace v plánovaném projektu, kde šlo o snímání a ukládání většího množství dat v průmyslovém prostředí z kamer Basler pomocí platformy LabVIEW. K provedení tohoto úkolu jsem nejprve potřeboval množinu *kodeků* a setříděnou množinu klasifikačních parametrů.

Množinu dostupných *kodeků* na daném PC poskytuje prostředí LabVIEW jednoduše pomocí svého API (*property node*). Jelikož na mém PC byly nainstalovány stejné moduly, které se standardně používají ve firmě ATEsystem (především šlo o NI Vision Development Module), mohl jsem si být téměř jistý, že k testování a pozdějšímu použití bude použita jednotná množina *kodeků*, za podmínky použití stejného OS.

Klasifikační parametry mi poskytnul zadavatel úkolu. V podobných projektech se jedná vždy minimálně o kvalitu, výpočetní náročnost, dobu trvání a výslednou velikost. Kvalita je na rozdíl od zbytku měřitelná většinou subjektivně. Dohromady tyto 4 parametry charakterizují daný *kodek* a danou PC sestavu a v různých případech jsou seřazeny dle různé priority. V mém případě šlo o tuto permutaci (od nejdůležitějšího): doba trvání, kvalita, výsledná velikost a výpočetní náročnost. Nyní se problém zredukoval na návrh a implementaci jednoduchého algoritmu s možností grafického výstupu v podobě grafů. Mým úkolem bylo vyřešit tento problém během 3 dnů.

Nejprve jsem připojil k PC kameru Basler a nastavil její parametry odpovídající prostředí laboratoře. Vyzkoušel jsem sejmout několik snímků, abych ověřil ideální nastavení a pokračoval jsem prací na kódu v LabVIEW. Algoritmus jsem navrhl tak, aby nejprve sejmul předem definované množství nekomprimovaných snímků z kamery, uložil je do operační paměti a následně pro každý dostupný *kodek* proběhla konverze a uložení na disk. Algoritmus rozlišoval *kodeky* pro ukládání videa (NI Vision Motion JPEG, Microsoft Video, FF Video, Cinepak, YUV 4:2:0 Planar) a samostatných obrázků (JPEG, JPEG2000, BMP, PNG, TIFF). Během konverze se monitorovalo průměrné využití všech procesorových jader (výpočetní náročnost), počítala se doba konverze, a nakonec se zjistila celková velikost výsledného souboru, resp. souborů. Nekompatibilní *kodeky* Intel IYUV a Y800 Uncompressed byly vyřazeny.

Testování probíhalo především na mém PC s procesorem Intel Core i5 M540 (2,53 GHz/2 jádra *Arrandale*) s SSD diskem Samsung 850 Pro (MLC) s OS Windows 7 (64 bit). Testovací data z kamery měla charakter 100 obrázků v rozlišení 2448x2048 ve formátu RGB (U32). Z grafů (viz Obr. 7) jsem získal veškerá potřebná data. V levém sloupci jsou parametry video *kodeků* (čas v sekundách, průměrná výpočetní náročnost v procentech

a velikost v MB), v pravém sloupci pak *kodeky* grafických formátů ve stejném pořadí. Po analýze dat a subjektivním ohodnocení kvality jednotlivých *kodeků* jsem určil výsledky.

Nejlepší *kodek* pro zpracování videa pro danou testovací sestavu a prioritu parametrů byl NI Vision Motion JPEG a pro zpracování obrázků JPEG (s kvalitou konverze nastavenou na maximum). Pokud by nevyhovovala kvalita JPEG, zvolil by se TIFF nebo BMP za cenu větší velikosti. Následující výčet představuje setříděné výsledky od nejlepšího po nejhorší:

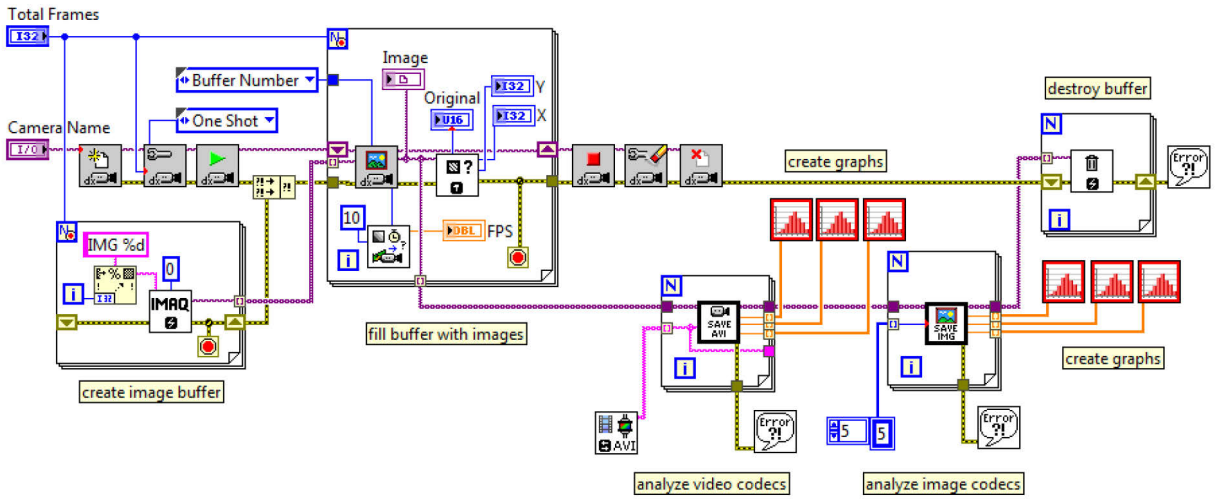
**Video *kodeky***

1. NI Motion JPEG (vítěz)
2. FF Video (značná velikost)
3. YUV 4:2:0 Planar (značná velikost)
4. Microsoft Video (nevyhovující kvalita)
5. Cinepak (extrémně dlouhá doba konverze)

**Grafické *kodeky***

1. JPEG (vítěz, kvalita nastavena na maximum)
2. TIFF (značná velikost)
3. BMP (značná velikost)
4. PNG (delší doba konverze)
5. JPEG2000 (extrémně dlouhá konverze, nastaven bezztrátový režim)

Aplikace *benchmark* pomohla kolegům z oddělení systémové integrace optimalizovat systém vizuální inspekce. Zdrojový kód včetně spustitelného souboru se stal součástí firemního *know-how*. Níže (viz Obr. 25) je nastíněna jedna z možných implementací aplikace.




Obr. 25: Ukázka možného způsobu, jak implementovat aplikaci *benchmark*

## 2.4 Testování

### 2.4.1 Zátěžové testy produktů

Během práce na vývoji FW a SW pro produkt „Proudový zdroj pro LED moduly“ došlo na situaci, kdy bylo třeba najít chybu, která se vyskytovala sporadicky (viz kapitola 2.1.1). Platforma LabVIEW cílí (mimo jiné) na podobný *use case*, kdy je třeba během co nejkratšího času vytvořit sofistikovaný testovací nástroj, což bylo mým úkolem. Během 1 dne jsem vytvořil aplikaci v LabVIEW jménem „Stress Test“ (v překladu: zátěžový test). Šlo o SW, který integroval LabVIEW ovladač proudového zdroje, a za pomocí jednoduchého algoritmu sloužil jako zátěžový tester. Aplikace pracovala ve dvou režimech, které šlo spustit nezávisle na sobě. Hlavní smyčka programu se opakovala, dokud ji uživatel manuálně nevypnul, během jedné iterace se pak buď provedl každý režim zvlášť, nebo oba za sebou.

První režim testoval veškerou funkcionalitu zdroje dle instrukční sady deklarované v katalogovém listu. Nejprve se nastavily veškeré podstatné parametry (*setpoint* proudu, limit proudu, limit napětí, úbytek napětí, režim regulace), pak se zapnul výstup. Určitou dobu se počkalo a pak se změřily veškeré dostupné veličiny (aktuální hodnota proudu, napětí, úbytek napětí, teplota, výkon, stavový registr) a nakonec se výstup vypnul.

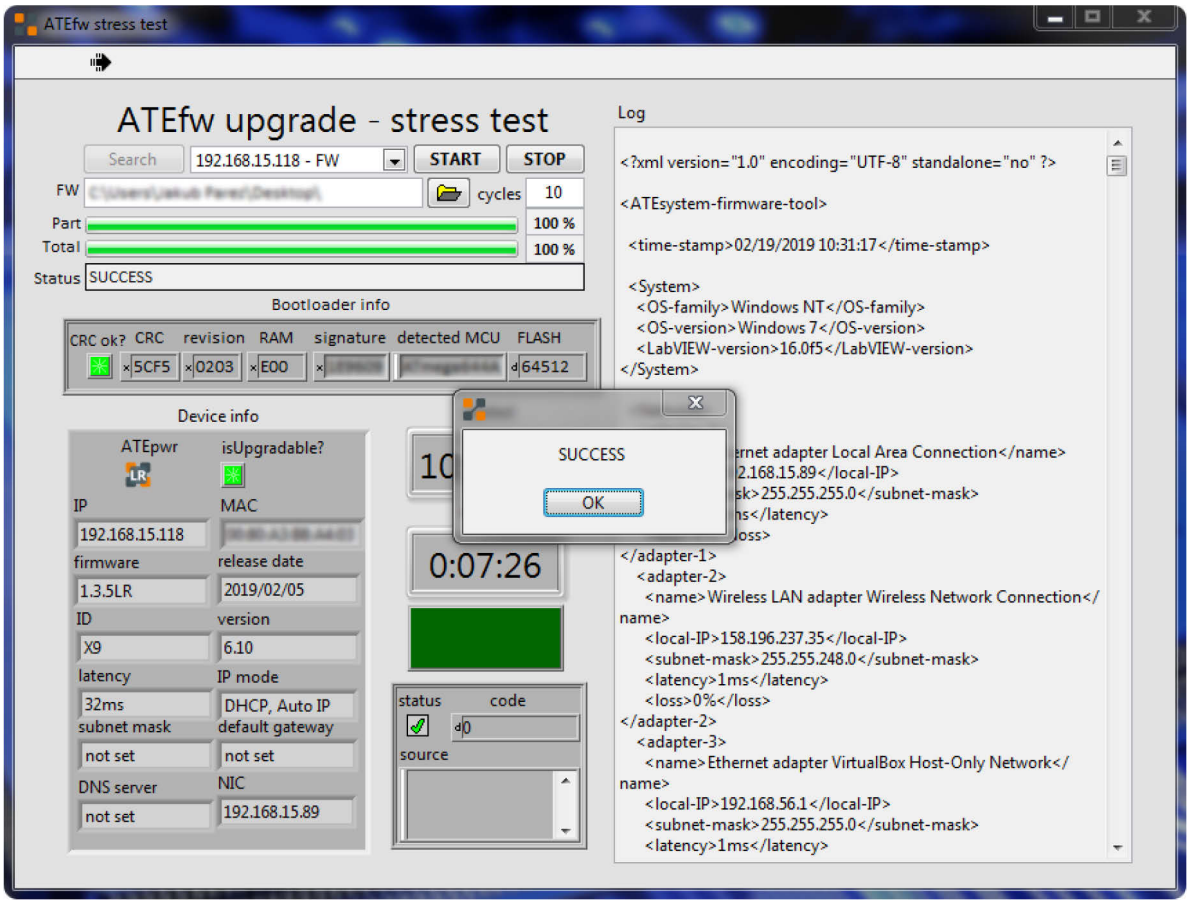
Druhý režim měl jiný úkol. Za pomocí datové struktury naplněné všemi validními instrukcemi (cca 70) a pseudonáhodného generátoru náhodných řetězců testoval kvalitu komunikace a korektnost implementace FW. Díky seznamu validních instrukcí tester věděl, kdy má zdroj odpovědět *OK* a kdy *ERROR*. Algoritmus šlo uživatelsky přizpůsobit, jednalo se například o typ generovaných znaků (čísla, malá písmena, abeceda, celá ASCII znaková sada), počet instrukcí, jejich délka nebo rychlost odesílání.

Tyto režimy se vykonávaly cyklicky a aplikace zobrazovala uživateli hlavní testované parametry v indikátorech a grafech na čelním panelu. Nejdůležitější indikátor byl samozřejmě množství chyb. Pokud došlo k chybě v některém z režimů, do průběžně generovaného *logu* se zapsala podrobná informace, zachycující aktuální stav, a inkrementovalo se počítadlo chyb. Poslední zajímavou volitelnou možností bylo propojit testovací řetězec s arbitrážním generátorem funkcí a testovat stabilitu zdroje pomocí zákmitů na napájení. Aplikace „Stress Test“ se ve vývojovém oddělení osvědčila. Kromě již diskutované chyby souběhu, která byla odhalena díky analýze stavové informace z *logu* testeru, je tento SW často používán pro ověření spolehlivosti přidaných funkcionalit v nových verzích FW.

Zátěžový test byl z podobných důvodů třeba provést u produktů „P-IRIS Controller“ a „Canon Controller“. Tyto produkty ze stejné kategorie sdílejí GUI i část kódu ovládací PC aplikace. Díky této podobnosti stačilo vytvořit jediný tester, ve kterém se bude pouze přepínat příslušný LabVIEW ovladač, který má každý produkt vlastní (instrukční sady se liší).

Hlavní rozdíl těchto řídících jednotek je v typu objektivu, který ovládají. „P-IRIS Controller“ je kompatibilní s objektivy, ve kterých se clona ovládá pomocí krokových motorů. Příkladem takového objektivu je například Theia TL1250P, který je standardně dodáván zákazníkovi s řídící jednotkou. Tento objektiv umožňuje kromě ovládání clony a ostření i ovládání přiblížení a vestavěného IR filtru, což není možné u objektivů Canon, kompatibilních s jednotkou „Canon Controller“. Tento fakt bylo třeba zohlednit v návrhu testeru.

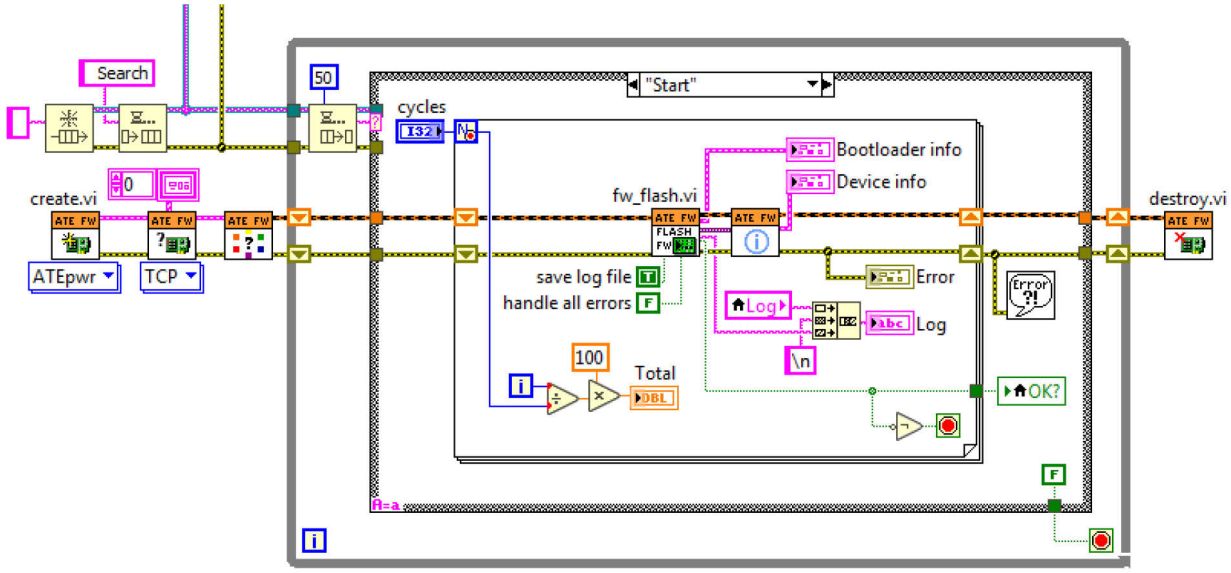
Během 1 dne jsem vytvořil aplikaci „Canon Controller/P-IRIS Stress Test“, která sloužila k zátěžovému testování zmíněných produktů. Princip byl velice podobný testeru proudových zdrojů. Cyklicky se prováděla předem definovaná sekvence, ve které se komunikovalo s řídící jednotkou. Testovaly se opět veškeré instrukce v různých kombinacích za různých podmínek a do *logu* se zaznamenávalo chování a stav testované soustavy. Vyhodnocení probíhalo analýzou *logu*. Práce probíhala rychleji, protože velikost instrukční sady proudového zdroje je zhruba 10x větší než velikost instrukční sady těchto jednotek. Ve výsledku SW fungoval korektně a splnil svůj účel.




Obr. 26: Čelní panel aplikace „Stress Test“ proudového zdroje, která testuje *bootloader*

Mým posledním úkolem bylo ověřit funkčnost nové funkcionality FW proudového zdroje. Jednalo se o vzdálený update FW přes *bootloader* a příslušnou ovládací aplikaci. *Bootloader* je malý kus kódu, který se nachází v paměti MCU v příslušné části k tomu vyhrazené. Bývá často psán v *assembleru*, protože jednak jde o kritický kód a jednak je kladen důraz na jeho velikost. Tento program pak zajišťuje nízko úrovňovou manipulaci s pamětí MCU, řízenou pomocí komunikace s externí aplikací. V případě proudového zdroje je použita komunikace pomocí TCP/IP, je tedy možné ovládat *bootloader* prakticky odkudkoliv. To je příhodné, pokud je třeba aktualizovat FW u zákazníka, který se nachází na druhé straně planety. Dnes bývá řešení zahrnující vzdálený update *de facto* již standardem.

Aplikaci jsem nazval „ATEfw Stress Test“ a princip její funkce spočíval v cyklickém provádění vzdáleného přehrání FW proudových zdrojů (viz Obr. 26). Jelikož byla ovládací aplikace napsána v LabVIEW, následná integrace s testerem trvala řádově desítky minut. Níže (viz Obr. 27) lze vidět část blokového diagramu testeru. Po zapnutí se nejprve zvolila IP adresa zdroje. Po připojení a nastavení parametrů se pak spustila cyklická sekvence, během které se na obrazovku vypisovaly stavové informace o průběhu nahrávání. Test proběhl úspěšně na několika nových i starších kusech, *bootloader* se tak stal součástí všech verzí FW (1.3.6 a vyšší) a aplikace „Device Configurator“ (1.0.0 a vyšší). Veškeré zmíněné aplikace pro zátěžové testování slouží k vývoji i servisu komerčních produktů, jsou proto určeny pouze pro interní firemní použití.




Obr. 27: Část blokového diagramu aplikace „Stress Test“, která testuje *bootloader*

# 3 Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe

V průběhu mé praxe jsem uplatnil mnoho znalostí a dovednostní získaných během bakalářského studia. Mou studovaný obor Počítačové systémy pro průmysl 21. stol je postaven moderně z půlky na elektrotechnických a z půlky na informatický dovednostech. Jelikož jsem první a letos jediný absolvent tohoto oboru, mohu podat nezaujatou objektivní zpětnou vazbu, a sice že se jedná o perfektní skladbu předmětů. Využil jsem samozřejmě znalosti matematiky a fyziky, které jsou v případě fakulty FEI nutnost. Základní principy měření elektrických veličin z předmětu Senzory a měření, základy elektrotechniky, kybernetiky a automatického řízení dynamických systémů se ukázaly jako nepostradatelný úvod do technické praxe. Výuka Virtuální Instrumentace (LabVIEW) byla na vysoké úrovni a poskytla mi pevné základy pro budoucí praxi. [14] Druhá část zaměřená na informatiku vynikala svou vzájemnou provázaností. V prvním ročníku absolvované předměty Algoritmy ([15]) a Programování (základy C++ a OOP) se ukázaly jako základní kámen, který některým kolegům z oboru Řídící a informační systémy chyběl. Výborný kurz o skriptovacích jazycích, především Pythonu, mi rozšířil obzory o úvod do funkcionálních jazyků a předmět Počítačové sítě spolu s Telekomunikačními sítěmi mi přijdou jako nutnost i pro „ne-síťaře“, jelikož znalosti zde nabyté používám téměř každý den.

# 4 Znalosti či dovednosti scházející studentovi v průběhu odborné praxe

Především mi scházely některé odbornější elektrotechnické znalosti, jelikož v mém oboru nebyl předmět Teorie Obvodů nebo podobný, musel jsem si některé konkrétní problémy z této oblasti nastudovat sám. Dále jsem byl v určitý moment konfrontován s projektovou implementací v *assembleru* 8bitové harvardské architektury, se kterou jsem neměl žádné předchozí zkušenosti, proto mi opět nezbývalo než si ji pečlivě nastudovat. Scházela mi také praxe týkající se projektově orientovaného vývoje, především v kontextu oddělení vývoje nových produktů, kdy za dobu trvání mého brigádnického vztahu s firmou jsem tento deficit významně potlačil.

# 5 Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení

Během mé bakalářské praxe jsem získal cenné zkušenosti z komplexního procesu vývoje HW i SW produktů. Podíval jsem se do průmyslových lokalit, kde jsem nasazoval svůj SW, který je součástí výrobního procesu v automobilovém odvětví. V laboratoři mi byla poskytnuta svoboda, kterou jsem využil jak při návrhu architektury, její implementaci a následném testování, kdy vedoucí nechal mnoho aspektů vývoje zcela na mě, tak při různých vedlejších projektech, které vznikly z mé iniciativy a postupem času se začlenily do hlavní produktové větve. Získal jsem komplexní znalosti vývoje FW a SW, zahrnující několik iterací fáze specifikace požadavků, analýzy a designu, implementace, testování a nasazení, kdy jsem mohl vidět, jak se teorie prezentovaná v knihách liší od reálné praxe, ve které hraje hlavní roli zákazník. Celou dobu mě těšilo příjemné, až rodinné prostředí firmy, kam jsem se vždy těšil, protože kolegové byli vždy velmi vstřícní, takže řešení libovolného problému vždy probíhalo přímočaře a profesionálně.

Výsledky mé práce ve firmě ATEsystem, s.r.o jsou firmware v jazyce C pro produkt „Proudový zdroj pro LED moduly“ ([6]), multiplatformní ovladač v jazyce C++ pro produkt „Řídící jednotka pro objektivy s krokovými motory“ ([11]), nástroj pro správu proudových zdrojů „Device Configurator“ v prostředí LabVIEW ([12]), tester animovaného blinkru světlometů osobních automobilů v LabVIEW, obslužná aplikace pro produkt „Řídící jednotka pro objektivy s krokovými motory“ v LabVIEW, *benchmark* grafických a video formátů v LabVIEW a zátěžové testery v LabVIEW pro zmíněné produkty („Stress Testy“).

# Literatura


[1] RIOS, B. *Security Evaluation of the Implantable Cardiac Device Ecosystem Architecture and Implementation Interdependencies*. WhiteScope, 2017 [online]. [cit. 2019-03-22]. Dostupné z: https://www.ledecodeur.ch/wp-content/uploads/2017/05/Pacemaker-Ecosystem-Evaluation.pdf

[2] VONDRÁK, Ivo. *Úvod do softwarového inženýrství.* Ostrava: VŠB – Technická univerzita Ostrava, 2012.

[3] NASA. *Mars Climate Orbiter Mishap Investigation Board Phase I Report.* [online]. [cit. 2019-04-07]. Dostupné z: https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf

[4] NCCIC. *Alert (ICS-ALERT-13-164-01) Medical Devices Hard-Coded Passwords.* Department of Homeland Security, 2013 [online]. [cit. 2019-04-07]. Dostupné z: https://ics-cert.us-cert.gov/alerts/ICS-ALERT-13-164-01

[5] *Katalogový list produktu Proudový zdroj pro LED moduly.* Materiál společnosti ATEsystem s.r.o. [online]. [cit. 2019-03-22]. Dostupné z: https://www.atesystem.cz/produkt/proudovy-zdroj-pro-led-moduly/

[6] *Uživatelský manuál produktu Proudový zdroj pro LED moduly.* Materiál společnosti ATEsystem s.r.o. [online]. [cit. 2019-04-07]. Dostupné z: https://www.atesystem.cz/produkt/proudovy-zdroj-pro-led-moduly/

[7] LEVESON, Nancy. *Medical Devices: The Therac-25* Washington: University of Washington, 1995 [online]. [cit. 2019-04-07]. Dostupné z: http://sunnyday.mit.edu/papers/therac.pdf

[8] *Katalogový list produktu Řídící jednotka pro objektivy s krokovými motory.* Materiál společnosti ATEsystem s.r.o. [online]. [cit. 2019-03-22]. Dostupné z: http://www.atesystem.cz/produkt/p-iris-controller-ridici-jednotka-k-objektivum-s-krokovymi-motorky

[9] VIRIUS, Miroslav. *Programování v C++: od základů k profesionálnímu použití.* Praha: Grada Publishing, 2018. ISBN 978-80-271-0502-1.

[10] CARVALHO, Marco, DEMOTT Jared, FORD, Richard, WHEELER David. *Heartbleed 101.* Florida Institute of Technology, 2014. [online]. [cit. 2019-04-07]. Dostupné z: https://profsandhu.com/cs5323_s18/heartbleed2014.pdf

[11] *Návod k použití ovladače produktu Řídící jednotka pro objektivy s krokovými motory.* Materiál společnosti ATEsystem s.r.o. [online]. [cit. 2019-04-07]. Dostupné z: http://www.atesystem.cz/produkt/p-iris-controller-ridici-jednotka-k-objektivum-s-krokovymi-motorky

[12] *Manuál k SW Device Configurator pro produkt Proudový zdroj pro LED moduly.* Materiál společnosti ATEsystem s.r.o. [online]. [cit. 2019-04-07]. Dostupné z: https://www.atesystem.cz/produkt/proudovy-zdroj-pro-led-moduly/

[13] BRESS, Thomas J. *Effective LabVIEW programming.* Allendale, N.J: NTS Press, National Technology and Science Press, 2013. ISBN 978-1-934891-08-7.

[14] BILÍK, Petr. *Virtuální instrumentace 2.* Ostrava: VŠB – Technická univerzita Ostrava, 2012.

[15] DVORSKÝ, Jiří. *Algoritmy 1.* Ostrava: VŠB – Technická univerzita Ostrava, 2007. [online]. [cit. 2019-04-07] Dostupné z: http://www.cs.vsb.cz/dvorsky/Download/SkriptaAlgoritmy/Algoritmy.pdf